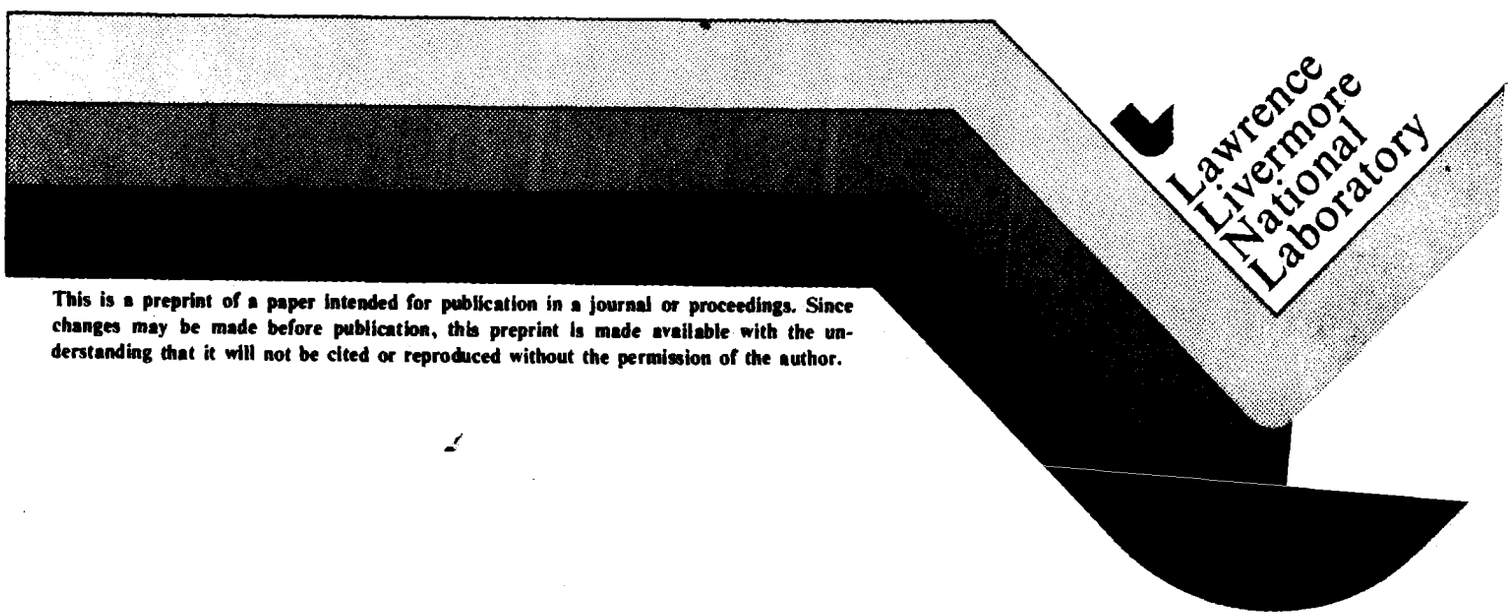
UCRL- 88765
PREPRINT

A Simulator for MIMD Performance Prediction-- Application to the S-1 MkIIa Multiprocessor

T. S. Axelrod
P. F. Dubois
P. Eltgroth

Computer Engineering

February 15, 1983



A Simulator for MIMD Performance Prediction--
Application to the S-1 MkIIa Multiprocessor*

T. S. Axelrod, P. F. Dubois, P. G. Eltgroth

Theoretical Physics Division
Mathematics and Statistics Division
Lawrence Livermore National Laboratory
Livermore, California

ABSTRACT

We describe a MIMD multiprocessor simulator and application of that simulator to a multiprocessor of current interest, the S-1 MkIIa. The simulator runs on the Cray-1 and is designed so that computational physics benchmarks are actually run and produce results. Simulator output from this run are fed into a second level (hardware) simulator which calculates the behavior of the multiprocessor. The simulator can simulate multiprocessors whose basic architecture is that of a few, large processors with or without data caches, sharing global memory through an interconnection switch.

The simulator is applied to investigate the behavior of 4 problems on the S-1: the benchmark physics code SIMPLE, a conjugate gradient linear algebra problem, a simple Monte Carlo problem, and a new method for neutron transport calculations.

*Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

I. INTRODUCTION

Our purpose in this paper is twofold. We begin by describing a simulator we have created for predicting the performance of realistic physics calculations performed on multiprocessors. We then describe four multiprocessor physics algorithms and present simulator results for their performance on a multiprocessor of current interest, the S-1 MkIIa [S-179]. The simulator is available for use on the LLNL, LANL, and MFECC systems, and a user's manual has been previously published [Axe82].

II. SIMULATION METHODOLOGY

1. Goals of the simulator

Simulation of computer systems is performed for a great variety of purposes. Among these are gate level simulations to verify logical correctness of a design, register level simulations which allow the effects of instruction sequences to be determined, and queuing models, which are most commonly employed to predict performance of computer systems under timesharing loads. The simulations described here fall in a less common category. Our goal is to predict the performance of an MIMD computer in solving large computational physics problems by some specified algorithm.

Our simulation has an additional goal of nearly equal importance. Since usable and accessible MIMD machines are still very rare, computational physicists, numerical analysts, and programmers have little or no experience with the "parallel world". Not only is a new set of performance related issues present, but there are new issues of logical correctness and choice of language constructs. MPSIM provides a readily available tool for gaining at least some of this experience. One crucial feature of the simulator is that

the algorithm being investigated is actually run in all its detail so that its numerical behavior (e.g stability, correctness of answers) can be observed. Among other things, this allows many bugs related to multiprocessing to be found.

2. The trace-driven two level methodology

Simulation under MPSIM occurs at two levels (MPSIM-1 and MPSIM-2). These two levels represent the software and hardware, respectively, of the integrated system being modelled. This two level structure was inspired by the work of L. Cox [Cox78]. At the first level of simulation we deal with autonomous instruction streams referred to as processes, without reference to any physical implementation, while at the second level we deal with physical processors. At present we assume that there is a one-to-one mapping between processes and processors, but this assumption is unnecessary, and affects only MPSIM-2. Thus extension of the simulator to include machines such as the Denelcor HEP [Smi78], which has multiple processes per processor, is at least in principle straightforward.

The two levels of the simulation (which can be run independently) are used for different purposes. At the first level of simulation, we are concerned primarily with the correct logical operation of a program, and the details of a particular multiprocessor implementation (such as the relative speeds of private and shared memory, the speeds of various synchronization operations, and so on) may be mostly ignored. On the other hand, the performance of a multiprocessor program may depend critically on the details of the implementation, and the second level of the simulation, which is driven by output from the first level, is intended to predict this performance. In

general, performance information will dictate changes in the program, which are incorporated by returning to the first level.

MPSIM-1 is an extended version of a simulator written by L. Sloan. It runs on the Cray-1 under CTSS and consists principally of a process scheduler and the machinery and data structures to save and restore process state information. The simulator provides the FORTRAN user with a number of subroutines which allow the creation and destruction of processes and implement a variety of synchronization operations. A brief summary of the available functions is given in Table 1. A complete description is available in [Axe82].

In operation the Level 1 simulator is a timesharing system in miniature. A single Cray-1 is multiplexed among the currently runnable processes and a simulated wall-clock is maintained. The level 1 simulator is in fact a simulation of a particular MIMD machine - a highly idealized multiple Cray-1 with both shared and local memory.

Clearly most of the characteristics of a parallel algorithm which will determine its performance on a real MIMD machine are contained in the details of its behavior on this idealized MIMD machine. Not only is the pattern of process creation, destruction, and communication present, but quite complete information is also available on the actual computation performed by each process, much of which is hardware independent. The linkage between the level 1 and level 2 simulator consists of abstracting the performance related information from the level 1 behavior and transferring it to level 2 where it may be interpreted in the context of a more realistic (and possibly quite different) MIMD machine.

What should actually be included in the information transmitted to level 2? One extreme approach would be to include the complete state of the

Table 1

Summary of MPSIM-1 Subroutines

Name	Purpose
MPINIT	initialize MPSIM-1
TRCINIT	begin trace output for MPSIM-2
FORK	start an additional process
FORKOFF	start multiple additional processes
SYNCAL	global synchronization barrier
PSEM	P operation on a semaphore
VSEM	V operation on a semaphore
PAWS	wake up the MPSIM-1 scheduler
JOINAL	terminate all but 1 process
PRCEND	terminate a process
TRCFIN	terminate trace output for MPSIM-2
MPFINI	terminate MPSIM-1 operation

simulated MP-Cray on every clock cycle. There are two serious problems with this, however. The quantity of information is several orders of magnitude too large to permit the behavior of a complete physics algorithm to be practically stored or processed. Even more serious, perhaps, is the fact that most of the information generated has no obvious relevance to machines other than a Cray-1 or a very close relative.

At the opposite extreme, one might simply count arithmetic operations performed by each process. This information is so incomplete, however, that the hardware model contained in Level 2 must of necessity be quite simple. In particular, details of interprocessor interactions which arise from memory conflicts and cache coherence problems cannot be modeled.

In general the choice of which information should be abstracted from the Level 1 behavior is partially subjective and must be based on a careful assessment of the characteristics of the system being modelled, and the size of the computational resources available for the simulation. The nature of the choice we have made for modelling the S-1 MkIIa is discussed in Section III. For the moment we merely note that it falls between the two extremes, preserving the details of each process' data referencing patterns and the arithmetic operations it performs, while neglecting many details of address calculations and instruction referencing patterns.

During the operation of MPSIM-1 the information needed by MPSIM-2 is gathered by machine instructions which have been automatically inserted into the user's object code. These instructions cause control to be temporarily transferred to simulator routines which write the desired machine state information to disc files, one for each process. This information gathering process is invisible to the user except for increased CPU charges.

MPSIM-2 views the events contained in each process' trace stream as requests for hardware services by a running process. Typically the services required are arithmetic operations and transfers of operands. Each request is satisfied as soon as possible within the constraints of the hardware model. We note that this may result in a time ordering for events in separate process streams that differs from that of MPSIM-1. Event ordering within a single process stream is preserved (unless the MPSIM-2 hardware model allows out-of-order execution), as are the interprocess orderings enforced by synchronization.

The output of MPSIM-2 is a detailed record of event histories for each of the simulated processors, which in practice are usually viewed in the form of graphical plots.

III. THE S-1 MkIIa HARDWARE MODEL

As the first application of our simulation techniques we chose to investigate the S-1 MkIIa multiprocessor. This machine is of great interest due to its imminent availability and supercomputer performance potential. Additionally, its design explores for the first time the use of cache memory in high speed multiprocessors.

In this section we describe the hardware model we have incorporated in SISIM-2 to model the S-1 MkIIa multiprocessor. When we created the model, the S-1 MkIIa implementation was far from complete, and a number of details were uncertain. This was particularly the case for the main memory, crossbar switch, and microcode used for interprocessor communication. Since hardware documentation was generally unavailable we have relied on conversation with S-1 project members [Far82] to fill in the gaps where possible. How

successful we have been in modelling the S-1 MkIIa as it is finally implemented will not be known until the machine is available for testing. We nonetheless are presenting the model and the results from it in the hope that it will be a generally useful contribution to performance assessment of MIMD machines.

1. Summary of S-1 hardware

The S-1 MkIIa multiprocessor [S-179,Far80,Far81] consists of up to 16 processors connected to a similar number of memory banks by a crossbar switch. Each processor is extensively pipelined and microcoded and possesses the following major resources:

1. Instruction cache (4k words)
2. Data cache (16k words)
3. Address arithmetic unit
4. Floating point adder
5. Floating point multiplier
6. 16 general purpose register files with 32 words each
7. Local memory (1M word class)

The S-1 pipeline is partitioned into two major units. The IBOX fetches and decodes instructions and handles all tasks associated with the fetching and storing of operands, including management of the cache, mapping of virtual to physical addresses, and memory protection. The ABOX is responsible for the remaining steps in instruction execution and calculates all results which will be stored in the register files or memory.

The processors are implemented with ECL 10K and 100K integrated circuits, while the local and global memories are implemented with 64kb MOS chips. The design cycle time of the processor is 50 ns for the instruction fetch and

decode unit (IBOX) and 25 ns for the arithmetic unit (ABOX). The word size is 36 bits.

The ABOX adder and multiplier are innovative in several respects [Far81]. They have been designed for low latency and are partitionable to allow calculation with operands of 18, 36, and 72 bits. Special attention has been devoted to achieving high speed on FFT's and other mathematical functions. In part for this reason the adder produces both the sum and difference of its operands simultaneously.

The instruction set is extensive, providing a wide variety of addressing modes, dyadic and triadic vector operations, and evaluation of mathematical functions among its most notable features.

Since the MPSIM-2 hardware model is driven by a trace stream obtained from a Cray-1, it is appropriate to contrast the S-1 MkIIa design with that of the Cray-1 [Cra76]. The most important differences include the following:

1. The design of the memory hierarchies differ greatly. The S-1 is a virtual address machine which utilizes a combination of fast ECL data and instruction caches, very large MOS main memory, and disk for demand paging. The Cray-1 is much simpler, relying on ECL technology for both its registers and 16 interleaved banks of main memory.

2. The S-1 has fewer functional units. All instructions must pass through either the adder or multiplier in the ABOX. On the Cray-1 there are thirteen functional units (including memory) which may execute instructions.

3. Vector operands on the S-1 are obtained directly from memory, while on the Cray-1 they are held in vector registers for use by the vector functional units. The vector stride must be 1 for the S-1, while it may be any value on the Cray.

4. The S-1 instruction set is more powerful than that of the Cray, so that multiple Cray instructions can often be replaced by a single S-1 instruction. The most common occurrence of this is in operand address computation, but there are many examples.

5. On the S-1, results are produced strictly in the order implied by the instruction order. On the Cray-1, although instructions are always issued in order, results may be produced out of order and may be produced by any of the functional units.

Both the number of instructions and the number of machine cycles devoted to address arithmetic are likely to differ greatly between the two machines. In favorable cases, however, the time taken to perform these operations is completely "hidden" through overlap with floating point operations and memory references. The same situation holds for conditional branches. Both machines are able to issue instructions at a maximum rate of one per cycle (12.5ns Cray-1, 50ns S-1 MkIIa). How closely this goal is approached is very sensitive to the optimization techniques employed by the compiler.

As sketched above, the S-1 MkIIa processor is highly complex. Our simulation models a small carefully chosen subset of its features. In most cases the model assumes that omitted features (e.g. prediction of conditional branches) work perfectly, so that the simulation results form an upper bound on performance, but there are exceptions.

In selecting the hardware features to be included in the model we began by recognizing that the effectiveness of the data cache will be a major determinant of performance. Occurrence of a data cache miss stops the IBOX pipeline until the required data can be obtained. Since main memory is much slower than the IBOX cycle time, data cache misses can easily impose a limit

on performance. This is especially the case when processors share data so that cache coherence problems arise. The model must therefore be able to determine the contents of the data cache of each processor at every stage of the computation.

On the other hand, we expect the effectiveness of the instruction cache to be generally high and of less importance in determining performance. This arises from both the generally much smaller size of total program instructions relative to cache size, and from the generally much higher localities observed for program instruction references compared to data references. This is most fortunate, since the instruction cache hit rate could not be modelled accurately without actual S-1 instruction streams for the computation. These will not be usefully available until a vectorizing compiler for the machine is completed.

Our model of the ABOX is quite simplified, since it ignores delays which result from data dependencies between operations. If all required operands are in cache the simulated ABOX is ready to execute a new instruction a fixed time (T_{issue}) after beginning the previous instruction. We have chosen T_{issue} to be the shortest possible - those obtained in the absence of data dependencies. This is not a necessary restriction on the model, since all data dependencies are in fact available in the trace stream. Again, our results are expected to form an upper bond on performance.

Most of the complexity of the S-1 processor arises from the need to keep arithmetic function units at the end of a long complex pipeline supplied with operands at a continuously high rate. (See [Kog81] and [Lor72] for good discussions of pipelined processors.) To achieve this, a variety of techniques is employed, including the partial decoupling of the IBOX and ABOX with an

operand queue, the prediction of conditional branches, and the prediction of values needed in address computations [Far81].

Our model assumes that in the absence of data cache misses, all of these techniques work perfectly, so that the functional units perform work at the maximum possible rate. Clearly this may be seriously in error for some computations. A Monte Carlo computation with a large number of quasi-random conditional branches is likely to run more slowly than our model would predict, for example, due to reduced effectiveness of the hardware conditional branch prediction strategy.

The model makes an additional simplifying assumption: the cost of address arithmetic instructions is ignored. As discussed above, this is equivalent to assuming that address arithmetic calculations are fully overlapped with other computations. Although the address arithmetic instructions themselves are filtered out, the memory reference instructions needed to fetch their operands are retained. This is necessary to include their effect on the data cache hit ratio (typically small).

The effect of the simplifications we have discussed so far is exclusively in the direction of predicting performance which is too high. The performance results for the Monte Carlo algorithm discussed in Section IV probably show these effects to a significant degree. The model, however, contains additional assumptions which work in the other direction. The most important of these is ignoring the chaining of vector operations on the S-1. Chaining allows triadic vector instructions to make simultaneous use of the adder and multiplier in the ABOX. This can increase the Mflop rate by up to a factor of 2. Measurements on the Cray-1, which has a more general chaining capability, show the effect to be somewhat less than this in most cases.

Of less importance is the fact that the model does not reflect the ability of the S-1 to calculate special functions (e.g. sin, log, exp) nearly as fast as multiplies. Measurements on the Cray-1, which calculates these functions slowly relative to multiply (120ns vs 12 ns per result in vector mode), shows that even physics simulation codes which make intensive use (by current standards) of special functions very rarely spend more than 10% of their time performing them. It is quite possible, however, that algorithms will evolve which exploit the S-1's ability to evaluate special functions inexpensively.

Table 2 list the Cray-1 instructions from the MPSIM-1 run which are included in the trace stream passed to MPSIM-2. The trace stream contains a four word record for each of these instructions which is executed by MPSIM-1. The record includes the identity of the process executing the instruction, the simulated cycle clock at instruction issue, and complete information on the registers and memory addresses utilized by the instruction.

Table 2

List of Cray-1 Instructions Passed to MPSIM-2

Cray-1	CAL		Description
034ijk	Bjk,Ai	,A0	Read (Ai) words to B register jk from (A0)
034ijk	Bjk,Ai	0,A0	Read (Ai) words to B register jk from (A0)
035ijk	,A0	Bjk,Ai	Store (Ai) words at B register jk to (A0)
035ijk	0,A0	Bjk,Ai	Store (Ai) words at B register jk to (A0)
036ijk	Tjk,Ai	,A0	Read (Ai) words to T register jk from (A0)
036ijk	Tjk,Ai	0,A0	Read (Ai) words to T register jk from (A0)
037ijk	,A0	Tjk,Ai	Store (Ai) words at T register jk to (A0)
037ijk	0,A0	Tjk,Ai	Store (Ai) words at T register jk to (A0)
060ijk	Si	Sj+Sk	Integer sum of (Sj) and (Sk) to Si
061ijk	Si	Sj-Sk	Integer difference of (Sj) and (Sk) to Si
061ijk	Si	-Sk	Transmit negative of (Sk) to Si
062ijk	Si	Sj+FSk	Floating sum of (Sj) and (Sk) to Si
062000	Si	+FSk	Normalize (Sk) to Si
063ijk	Si	Sj-FSk	Floating difference of (Sj) and (Sk) to Si
063i0k	Si	-FSk	Transmit normalized negative of (Sk) to Si
064ijk	Si	Sj*FSk	Floating product of (Sj) and (Sk) to Si
065ijk	Si	Sj*HSk	Half precision rounded floating product of (Sj) and (Sk) to Si
066ijk	Si	Sj*RSk	Full precision rounded floating product of (Sj) and (Sk) to Si
067ijk	Si	Sj*ISk	2 - Floating product of (Sj) and (Sk) to Si
070ijx	Si	/HSj	Floating reciprocal approximation of (Sj) to Si
10hijkm	Ai	exp,Ah	Read from ((Ah) + exp) to Ai (A0-0)
100ijkm	Ai	exp,0	Read from (exp) to Ai
100ijkm	Ai	exp,	Read from (exp) to Ai
10hi000	Ai	,Ah	Read from (Ah) to Ai
11hijkm	exp,Ah	Ai	Store (Ai) to (Ah) + exp (A0-0)
110ijkm	exp,0	Ai	Store (Ai) to exp
110ijkm	exp,	Ai	Store (Ai) to exp
11hi000	,Ah	Ai	Store (Ai) to (Ah)
12hijkm	Si	exp,Ah	Read from ((Ah) + exp) to Si (A0=0)
120ijkm	Si	exp,0	Read from exp to Si
120ijkm	Si	exp,	Read from exp to Si
12hi000	Si	,Ah	Read from (Ah) to Si
13hijkm	exp,Ah	Si	Store (Si) to (Ah) + exp (A0=0)

Table 2 (Continued)

Cray-1	CAL		Description
13Uijkm	exp,	Si	Store (Si) to exp
13hiUUU	,Ah	Si	Store (Si) to (Ah)
140ijk	Vi	Sj&Vk	Logical products of (Sj) and (Vk) to Vi
141ijk	Vi	Vj&Vk	Logical products of (Vj) and (Vk) to Vi
142ijk	Vi	Sj!Vk	Logical sums of (Sj) and (Vk) to Vi
142i0k	Vi	Vk	Transmit (Vk) to Vi
143ijk	Vi	Vj!Vk	Logical sums of (Vj) and (Vk) to Vi
144ijk	Vi	Sj/Vk	Logical differences of (Sj) and (Vk) to Vi
145ijk	Vi	Vj/Vk	Logical differences of (Vj) and (Vk) to Vi
145iii	Vi	0	Clear Vi
146ijk	Vi	Sj!Vk&VM	Transmit (Sj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
146i0k	Vi	#VM&Vk	Vector merge of (Vk) and 0 to Vi
147ijk	Vi	Vj!Vk&VM	Transmit (Vj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
150ijk	Vi	Vj<Ak	Shift (Vj) left (Ak) places to Vi
150ij0	Vi	Vj<1	Shift (Vj) left one place to Vi
151ijk	Vi	Vj>Ak	Shift (Vj) right (Ak) places to Vi
151ij0	Vi	Vj>1	Shift (Vj) right one place to Vi
152ijk	Vi	Vj,Vj<Ak	Double shift (Vj) left (Ak) places to Vi
152ij0	Vi	Vj,Vj<1	Double shift (Vj) left one place to Vi
153ijk	Vi	Vj,Vj>Ak	Double shift (Vj) left (Ak) places to Vi
153ij0	Vi	Vj,Vj>1	Double shift (Vj) left one place to Vi
154ijk	Vi	Sj+Vk	Integer sums of (Sj) and (Vk) to Vi
155ijk	Vi	Vj+Vk	Integer sums of (Vj) and (Vk) to Vi
156ijk	Vi	Sj-Vk	Integer differences of (Sj) and (Vk) to Vi
156i0k	Vi	-Vk	Transmit negative of (Vk) to Vi
157ijk	Vi	Vj-Vk	Integer differences of (Vj) and (Vk) to Vi
160ijk	Vi	Sj*FVk	Floating products of (Sj) and (Vk) to Vi
161ijk	Vi	Vj*FVk	Floating products of (Vj) and (Vk) to Vi
162ijk	Vi	Sj*HVk	Half precision rounded floating products of (Sj) and (Vk) to Vi
163ijk	Vi	Vj*HVk	Half precision rounded floating products of (Vj) and (Vk) to Vi
164ijk	Vi	Sj*RVk	Rounded floating products of (Sj) and (Vk) to Vi
165ijk	Vi	Vj*RVk	Rounded floating products of (Vj) and (Vk) to Vi
166ijk	Vi	Sj*IVk	2 - floating products of (Sj) and (Vk) to Vi
167ijk	Vi	Vj*IVk	2 - floating products of (Vj) and (Vk) to Vi
170ijk	Vi	Sj+FVk	Floating sums of (Sj) and (Vk) to Vi
170i0k	Vi	+FVk	Normalize (Vk) to Vi
171ijk	Vi	Vj+FVk	Floating sums of (Vj) and (Vk) to Vi
172ijk	Vi	Sj-FVk	Floating differences of (Sj) and (Vk) to Vi
172i0k	Vi	-FVk	Transmit normalized negatives of (Vk) to Vi
173ijk	Vi	Vj-FVk	Floating differences of (Vj) and (Vk) to Vi
174ij0	Vi	/HVj	Floating reciprocal approximations of (Vj) to Vi
174ij1	Vi	PVj	Population counts of (Vj) to Vi

Table 2 (Continued)

Cray-1	CAL		Description
174ij2	Vi	QVj	Population count parities of (Vj) to Vi
175xj0	VM	Vj,Z	VM=1 where (Vj) = 0
175xj1	VM	Vj,N	VM=1 where (Vj) .NE. 0
175xj2	VM	Vj,P	VM=1 where (Vj) positive
175xj3	VM	Vj,M	VM=1 where (Vj) negative
176ixk	Vi	,A0,Ak	Read (VL) words to Vi from (A0) incremented by (Ak)
176ix0	Vi	,A0,1	Read (VL) words to Vi from (A0) incremented by 1
177xjk	,A0,Ak	Vj	Store (VL) words from Vj to (A0) incremented by (Ak)
177xj0	,A0,1	Vj	Store (VL) words to Vj from (A0) incremented by 1

3. Details of S-1 Mark IIa Hardware Model

The S-1 Mark IIa is assumed to be a multiprocessor in which external communication initiated by a processor is carried out through a crossbar switch. The communication can be directed either to another processor or to a global memory bank. Each processor has a private cache memory and a private local memory. The address space of the individual processor is quite large, so that the global and local memories are not effectively limited in size. The cache memory for a processor allows 4 reads and 4 writes per 50ns IBOX cycle. The cache is set associative, that is, a memory location belongs to a given set determined from its address. There are 256 possible sets for the S-1 Mark IIa and there are four lines of data from a given set which may be in cache memory at a time. A line of data consists of 16 contiguous 36 bit words. From the above it can be seen that the design size of cache memory is 256 x 4 x 16 = 16K words. The simulator treats all but one of the specific numbers above as parameters which can be changed to investigate the effects of new technology and design changes. The exception is the line size, fixed at 16 words. A complete list of the default parameter values is given in Table 3.

Other parameters are set at run time by the user. The number of processors may be chosen to be 1 to 16. The number of global memory banks may also be chosen to be 1 to 16. The equivalent precision of the machine may also be set to single or double precision. In double precision two 36 bit words are used to represent the information originally present in one 64 bit CRAY word. This is the default simulator setting.

Memory addresses passed to MPSIM-2 are treated as virtual addresses. The mapping of virtual to physical addresses is assumed to result in successive blocks of virtual addresses being mapped to successive global memory banks. This type of interleaving was chosen because it results in balanced loading of the memory banks, but it does not always result in optimal performance. The block size may be set by the user to be any power of 2, with 2**10 = 1024 words being the default value. This is the smallest block size permitted by the S-1 architecture.

If all required operands are in cache an ABOX operation begins at a time T_{issue} after the previous operation was begun. If any required operands are not in cache, the ABOX must wait until they can be obtained. The issue time for a scalar arithmetic operation is two cycles, independent of operation type. Issue time for a vector operation is VL+1 cycles for single precision and 2*VL+1 cycles for double precision, where VL is the vector length.

Instructions are assumed to be available for execution immediately after the previous instruction completes. Thus most functions of the I-Box are not simulated, as discussed earlier in Section III. Chaining is not simulated.

The S-1 has vector instructions which can accept inputs from memory and return outputs to memory with no need for intervening register storage. Thus the S-1 has no vector registers. If a trace is generated on a machine which does have vector registers (such as the Cray 1), references to those vector registers are treated as references to local memory by the S-1. Since the vector registers are usually heavily used, the S-1 equivalents will almost always be in cache. A transfer between a vector register equivalent in cache and a memory location in cache incurs no time charge since realistic S-1 coding

would have no such transfer. It should be realized that the use of "vector registers" in this sense is advantageous on cached-based machines such as the S-1. The vector registers on the Cray-1 are frequently utilized by the compiler to hold intermediate vector results within a vector loop. If such temporary operands are each assigned unique locations in memory, many unnecessary cache misses result.

The basic memory reference operation in the S-1 Mark IIa is the read operation. Every write instruction which does not have the result datum in cache with write access is implemented by first carrying out a read instruction to bring the datum into cache. The write operation takes place in cache and the result is left there until some other processor requests it or the cache line is needed for another operation.

When a memory reference is to be made, the address is converted to a physical address and the set number is generated. The address falls within a line having that set number. If the line is currently active, then a cache miss is immediately counted and the memory reference is recycled for another attempt later. The conditions under which a line is considered to be active are:

a. The line is to be transferred to or from global memory and write access has been or will be granted.

b. The line is to be transferred from global memory with read access and write access is desired by the current processor.

c. The line is in another cache with read access and write access is desired by that other processor.

Next the cache is checked to see which element of the set (if any) has the desired line. The locus of the original datum - whether global or local - is also determined. If the line is in cache and is valid then we have a possible cache hit. A valid local variable in cache is a cache hit. A valid global memory line in cache is a cache hit unless write access is desired but not yet granted. All memory references for a given operation can be completed in one cycle if all are cache hits.

Cache misses will involve the crossbar switch network for anything but a local variable. A line in cache with read access granted but write access desired gives rise to a check of all other processors and invalidation of that line for those that have read access. If a line must be replaced in cache, the set is scanned for already invalid lines and, if none are found, the least recently used line in the set is marked. If write access has been granted for that line, it will be written out to the appropriate memory after the new line has been brought in.

If a line in global memory is desired for read access and no other processor has it with write access, the appropriate switch path is requested and the line delivered when the path is clear and the proper time has elapsed. The same is true if the line is desired for write access and no other processor has the line in cache. Once a switch path has been opened, it is held open until that transaction is completed.

If a line in global memory is desired and any other processor has write access to that line, then a message is passed through the switch requesting the line to be written back to global memory. After the line is written back it is read by the original processor.

If a line in global memory is desired for write access and other processors have it in cache with read access, messages are passed through the switch to invalidate the other cache copies. Then the line is obtained with write access from global memory.

The switch in the model is assumed to be of the crossbar type. The crossbar switch has input and output lines which are used to establish the paths of communication. A processor may initiate communication either to another processor or to a global memory bank. In the usual mode of operation, it is assumed that a memory bank and a processor having the same number will share a switch output line. Thus switch conflicts would occur whenever the two possible outputs are referenced at the same time. This mode has been used for the results described in Section IV. An alternate description of the crossbar switch assigns memory banks and processors to separate switch output lines. This latter mode has no switch conflicts between memory banks and processors. If either input or output line is busy when a new request is made of the switch, the request is placed into a queue to await service.

This model of the S-1 Mark IIa places all requests for a switch path and routing decisions at the switch level. In the actual machine, a given processor may hold a request for a switch line until it can be issued. This should make no significant difference in results, since we are really simulating the same queuing procedure.

The queuing procedure for the crossbar switch has two levels, here called active and pending. In each level only one message from a given processor may be present at a time. The switch accepts messages from the top of the active queue, which is filled from the bottom with messages from the top of the pending queue. This guarantees that, if two processors are competing for the same resource, one processor cannot be served twice before its competitor is served.

The pending queue is filled on a first come-first serve basis. Overflow messages are held in a special storage area and submitted to the pending queue on a first come-first serve basis. This is done in lieu of forcing the processor to resubmit requests to the queuing structure.

Table 3

S-1 Mark IIa Machine Characteristics and Simulator Default Values

Attribute	Default Value
Number of sets in cache	256
Number of lines from a set in cache	4
Number of words in a line	16
Number of bits in a word	36
Time - basic arithmetic cycle	25 ns
Time - to produce one result from a single precision vector instruction with values in cache	25 ns
Time - to produce one result from a double precision vector instruction with values in cache	50 ns
Time - data request and transfer from global memory to cache	2000 ns
Time - data transfer from cache to global memory	2000 ns
Time - data request and transfer from local memory to cache	1000 ns
Time - data transfer from cache to local memory	1000 ns
Time - between successive services from a global memory bank to cache	1300 ns
Time - between successive starts of data transfers from cache to global memory banks	1700 ns
Time - to start invalidation search for a given line in other cache memories	150 ns
Time - to send a message to change validity of a line in another processor's cache memory	300 ns
Time - to finish invalidation search for a given line in other cache memories	150 ns
Time - to invalidate a line in cache after message received	125 ns

In order to run problems in the parallel processor simulation, some functions of the operating system had to be modeled. Specifically, parallelization constructs such as forks, join, semaphores, etc. were assumed to be handled by the operating system. The basic philosophy taken was that the operating system was very efficient--able to carry out its assignments in a minimum time. One cycle was charged to account for each simulator operation. A charge of zero cycles leads to unrealistic situations with strings of simulator operations being carried out instantaneously/simultaneously. The time charges incurred for the operating system are certainly less than those which would occur for a real system!

The functions supported by the operating system are

1) Start or restart a processor.
2) Fork off a task to another processor.
3) P a semaphore (Decrement by one. If result less than zero processor will wait).
4) V a semaphore (Increment by one. If result less than or equal to zero, choose a waiting processor at random and command it to resume execution).
5) Terminate a process on a processor.
6) Synch All. When each processor reaches a synchronization point, it will wait. When all have reached, all will be restarted.
7) Join All. When each processor reaches a join point, it will wait. When all have reached, one specified processor will restart.

4. Discussion of verification

There are two separable verification issues associated with the simulation of computers as we are performing it. The first of these is how well the model reflects the reality of the machine being simulated, while the second is whether the simulator as implemented embodies the chosen model without errors. The first question is nearly impossible to answer in the absence of a functioning machine, compiler, and a broad spectrum of comparison tests. On the other hand, the second question may be answered with some confidence through detailed analysis of simulation output and comparison with analytic models which can be constructed for special cases.

An analytic model can be constructed for a highly idealized, but nonetheless instructive, situation. We consider the case in which each processor is executing an instruction stream that has no dependencies on any other processor's stream (they share no data and perform no synchronization). The instructions consist only of dyadic arithmetic operations on long vectors, which are assumed to reside in global memory. This situation is closely approximated when a vectorized sweep over a large two-dimensional mesh is partitioned by assigning each processor a section of the mesh disjoint from that of the others.

The model is described by the following parameters:

- P -- Number of processors
- M -- Number of memory banks
- T_a -- Time between successive arithmetic results in vector mode
- T_m -- Time to transfer a line between data cache and a memory bank in the absence of bank conflicts
- i -- Vector stride (address increment between successive operands)
- l -- Number of operands in a cache line

When performing a vector operation the S-1 MkIIa does not prefetch operands, so that when a cache miss occurs the ABOX stops producing results until the miss is serviced. The time to produce N results is thus

$$T = N*T_a + K*T_M$$

where K is the number of cache misses that occur and T_M is the average time to service a cache miss (including memory conflicts). When N is sufficiently large relative to the cache size the first reference to a given memory line within any of the vector operands will always result in a cache miss. Each memory line is referenced a total of [1/i] times by the vector operation, however, and all but the first of these are cache hits. Since the vector operation makes 3N memory references the number of cache misses is then

$$K = 3N/\ [1/i]$$

The rate at which results are produced is then

$$s = N/T = 1/(\ T_a + 3T_M/[1/i]\ )$$

$$= s0/(\ 1 + 3/[1/i]*(T_M/T_a)\ )$$

where $s0=1/T_a$ is maximum vector speed of the machine, which is achieved only in the complete absence of cache misses.

To complete the model we need to know T_M. The average miss service time is increased over T_m, the memory access time, by two effects - the presence of memory conflicts and the need to write back to memory cache lines which have been modified by the vector operation. We note that here "memory access time" is the total time the ABOX is stopped due to a cache miss, and includes time for microcode execution by the processor and for transfer of data through the crossbar network in addition to memory bank access time.

Estimating the effect on performance of writing back LRU'd cache lines to memory is difficult. The number of such writebacks is simply K/3, since the

two input operands are unchanged and only lines of the result operand need be written back. This increases the number of cache line transfers to 4K/3. The ABOX is not stopped by the writeback, however, unless another cache miss occurs. This reduces the impact of writebacks on performance so that it is perhaps a 20% effect. For present purposes we ignore it, although the simulator should model it accurately.

Yen et al [Yen82] have recently published a simple model which accurately predicts the effects of conflicts on T_M. Their model assumes that processors access memory banks randomly. When a requesting processor finds a memory bank busy it resubmits its request on the next memory cycle. With this model we have that

$$T_M = T_m/f(P,M,\psi)$$

where ψ is the probability that a processor issues a memory request in a memory cycle time in the absence of conflicts and f results from solving a nonlinear algebraic equation. For our situation

$$\psi = K*T_m/(K*T_m + N*T_a) = 1/(1 + T_a/T_m*[1/i]/3)$$

and this allows the speed, s, to be calculated.

To compare our simulator with this analytic model we wrote an application problem in which each of P processors performs a long vector add on operands disjoint from those of the other P-1 processors. The cache is initially empty and the vector length, N, was chosen to be 5000, just short enough that LRU writebacks do not occur. The page size was decreased to 16 words, and, as in all our runs, successive pages are located in successive memory banks. This should closely duplicate the case for which the model is valid.

Table 4 shows a comparison of the results of the MPSIM simulator and the analytic model. For all cases T_a=25 ns and T_m=2000 ns, so that the basic

speed of each processor is s0=40 Mflops. The speeds in the table are expressed in Mflops and are per processor.

Table 4

P	M	i	Hit ratio	ψ	f	s(MPSIM)	s(analytic)
4	2	1	.9375	.938	.449	1.3	1.2
4	4	1	.9375	.938	.668	2.2	1.7
4	4	4	.7500	.984	.648	0.5	0.4
4	4	16	0.	.996	.643	0.1	0.1
4	8	1	.9375	.938	.829	2.2	2.1
8	8	1	.9375	.938	.651	1.9	1.7
16	16	1	.9375	.938	.642	1.6	1.6

The agreement of the model and the simulator is quite good over a wide range of parameters. That the simulator predicts somewhat better performance is expected, since the simulator recognizes that some machine operations included in the time T_m are not susceptible to conflicts, while the model does not. This agreement increases our confidence that the simulator is working as intended. To check cases not addressed by this test, such as proper handling of cache writebacks and cache coherence operations we have analyzed many instruction sequences in detail.

We note that in all cases the attained performance is far below s0. Instead of being limited by the speed of the floating point functional units,

performance is limited by memory bandwidth. This is also the case for the algorithms we discuss in Section IV.

IV. FOUR MULTIPROCESSOR PHYSICS ALGORITHMS

1. General comments on partitioning and programming strategy.

The four algorithms we discuss below share a general approach to the partitioning of problems into multiple computational tasks and the management of the parallel processes which perform them. We have in general taken an existing uniprocessor algorithm and adapted it for a MIMD machine with minimal changes to its data and control structures. Our view is that this is a necessary, but far from final, step in algorithm development for MIMD machines.

All of the algorithms we discuss are iterative. The computation consists of a series of "cycles", each nearly identical in behavior. We have preserved this structure in the MIMD versions by requiring global synchronization at the end of each cycle. Additional global synchronizations are required within each cycle to achieve logical correctness while retaining the basic control structure of the uniprocessor version.

The partitioning of the problems is based on some simple static partitioning of the problem's data structures. In the case of two-dimensional meshes, for example, each parallel process works with a fixed contiguous group of rows or columns. We have also confined ourselves to a fixed one-to-one mapping of processes to physical processors, so that the effects of process scheduling strategies could be initially ignored.

In most cases the first MIMD algorithms we produced had unacceptably low levels of predicted performance. A few simple techniques were applied to achieve higher performance. Most commonly, additional processor-local copies of global data were supplied, or loops were reordered to reduce cache misses.

The algorithms we present are far from being optimally tuned for the S-1, however. Significantly increased performance may be obtained in some cases by taking advantage of the interprocessor message hardware.

The simulator output includes a large number of performance diagnostic quantities. In our discussion of algorithms we have used the following definitions. The cache hit ratio is the ratio of the number of memory references originally finding a datum in cache divided by the total number of memory references. Efficiency is the ratio of the time one processor needs for a calculation divided by P times the time it takes P processors to finish the same calculation. Traffic ratio is the ratio of the number of bits transferred between cache and other memory to the total number of bits which flow between cache memory and its CPU.

Speed is measured in megaflops (MFLOPS), millions of floating point operations performed per second. Speedup is the ratio of the time it takes one processor to perform a calculation divided by the time it takes a specified number of processors to complete the calculation.

It is important to notice that direct comparison of MFLOPS on a parallel machine versus a sequential machine may be misleading since many parallel algorithms perform significantly more arithmetic operations than their sequential analogs. For the algorithms we discuss below, however, such redundant operations are completely negligible.

2. SIMPLE

SIMPLE [Cro78] is a widely distributed code which models the hydrodynamic and thermal behavior of fluids in two dimensions. The hydrodynamics is a standard Lagrangian formulation using an artificial viscosity. Heat transfer

is performed in the diffusion approximation using a single ADI iteration on a five point implicit difference operator. Thermodynamic properties of the fluid are obtained by table lookup and biquadratic interpolation between table entries.

After an initialization phase the calculation consists of a sequence of timesteps each of which advances the solution by a time increment DT in the following manner:

1. calculate the pressure in each zone given the temperature and density
2. compute the acceleration, new velocity and new position of each zone
3. compute new zone volumes
4. compute the artificial viscosity and Courant timestep limit for each zone
5. calculate new zone internal energy after hydrodynamic work. To maintain sufficient accuracy the new energy is first predicted using old thermodynamic quantities. The predicted energy is then used to calculate more accurate thermodynamic quantities which are used for the final calculation of the new internal energy.
6. calculate new temperature after hydrodynamics from new density and internal energy
7. calculate heat diffusion coefficient for each zone
8. calculate the coupling constants for the column direction (one per zone)
9. calculate an intermediate temperature by solving a tridiagonal linear system which couples zones in the same column and has the temperature from step 7 as a right hand side.
10. calculate the coupling constants for the row direction (one per zone)

11. calculate the new temperature by solving a tridiagonal linear system which couples zones in the same row and has the temperature from step 9 as a right hand side.

12. calculate a heat diffusion DT for each zone from the rate of change of its temperature

13. calculate new zone internal energy from new temperature

14. calculate whole problem sums (kinetic and internal energy and heat flow across problem boundaries) and next DT by finding the minimum over the entire mesh of the zonal Courant and heat diffusion DT's

Although space does not permit a complete discussion of these calculational steps, some comments are in order. Steps 1 through 6 constitute the hydrodynamics portion of the timestep. The method is explicit, and the new values for a zone depend only on the previous values of that zone and its six nearest neighbors. Steps 7 through 13 constitute the heat conduction portion of the timestep. The method is implicit, and the new temperature of a zone depends on the previous values of all zones in the mesh. This difference is quite important for a multiprocessor. It is also important to note that boundary zones require special treatment for both hydrodynamics and heat conduction and require more calculations than interior zones.

We began with a version of SIMPLE which is almost completely vectorized by the CFT or CIVIC compilers for the Cray-1. This program was modified for a multiprocessor in a straightforward manner. Each processor is assigned a fixed contiguous group of mesh columns for which it is responsible at each stage of the calculation. With the exception of the heat conduction row sweep (discussed below) all calculations are vectorized along columns. All arrays are stored columnwise, so this results in vector operations with unit stride,

which is ideal for the S-1. Synchronization barriers were emplaced between calculation stages when required to ensure the proper data dependencies. Fourteen such barriers are required within the timestep loop. The program contains a single critical section implemented with semaphores which is used for updating scalars which depend on the global mesh (step 14).

With the exception of a single processor which is given the additional duty of performing output to the edit file, all processors execute identical code. In addition to the main data structures of the problem, which are in shared memory, each processor is supplied with local data structures which are stored in processor private memory. A few of these, such as loop indices, must be supplied to ensure logically correct operation. The majority of them, however, are made local to increase performance. These include the material property tables, arrays holding the coupling coefficients for heat conduction, and scratchpad arrays used for holding temporary results. Local scalars are used to hold each processor's contribution to global mesh quantities such as total kinetic energy and minimum timestep (step 14).

As mentioned above, the heat conduction calculation contains an exception to the column group partitioning used in the remainder of the code. The heat conduction step in fact raises some interesting partitioning issues, and deserves special discussion. Since the value of a zone temperature after the heat conduction step depends on the previous temperatures of all mesh zones, it is inevitable that this calculation on a multiprocessor will involve substantial interprocessor communication. There are at least two different ways of organizing the calculation.

1. Straightforward partitioning. During the column sweep each processor is given a group of columns, while during the row sweep each processor is

given a group of rows. All interprocessor communication is handled invisibly by the cache coherence algorithm in the hardware.

2. Wavefront with row blocking. Each processor works with a group of columns for both the row and column sweep. During the row sweep processor i+1 cannot begin on its portion of a row until processor i is finished with its portion of that row.

The second method, which is advocated by Gilbert [Gil79] attempts to reduce interprocessor communication demands at the expense of reduced parallelism and increased program complexity. The idea is that each processor "owns" the data associated with a column group. During the row sweep a processor continues to work with this local data except at the boundaries of its column group, where interprocessor communication is required. For large column groups the relative cost of this communication becomes small.

This strategy is effective only if the data associated with a column group remains local to a processor throughout the calculation. On the S-1 MkIIa there are two kinds of local data - that which is contained in the data cache and that which is contained in processor private memory external to the cache. Data which is in cache remains there only until it is removed by the replacement algorithm to make way for other data. This kind of locality is transient and for large column groups is destroyed during the column sweep. For the wavefront algorithm to work as intended, therefore, the column group data must be held in processor local memory. Processors utilize the crossbar switch only to send data packets directly to other processors.

The wavefront approach therefore is a form of "distributed computing" and as such requires quite different programming constructs than employed in uniprocessor scientific programs. In particular, the extended version of

FORTRAN we are using has no convenient facilities to distinguish local from shared data or for the sending and receiving of interprocessor messages. The first approach, however, allows the heat conduction part of SIMPLE to be programmed in the same style as the rest of the code.

For the simulations reported here we have chosen to use the straightforward partitioning appropriate to a tightly coupled approach to multiprocessing. It is interesting to note that for large problems the overhead directly attributable to interprocessor communication still becomes small. It is of course true that each processor writes out many cache lines to main memory which are later read by other processors. The point is that the vast majority of these reads and writes are performed by the normal cache replacement algorithm, and would occur even in the absence of other processors. The penalty of the shared memory approach is then paid mainly in bank conflicts.

We have run problems varying in size from 20C,20R to 90C,40R (where C denotes number of columns and R number of rows) and utilizing from 1 to 16 processors. Most of these have been run in single precision mode, although a few double precision runs have also been made. The results we report here are for a single execution of steps 1 - 14 of the timestep advance. Runs with multiple cycles have been performed, and show that the transient effects from starting with an empty cache are quite small. Table 5 summarizes our results for a 90C,20R problem run in single precision with varying numbers of processors.

Table 5

P	Mflops	Speedup	Efficiency
1	9.04	1.00	1.00
2	16.00	1.77	.89
4	26.45	2.93	.73
6	32.17	3.56	.59
8	34.7	3.84	.48

The efficiency drops rapidly for P>4 and there is clearly little to be gained by running this problem on more than 8 processors.

Figures 1 through 5 show detailed simulator results for a single run, a 90C,20R single precision problem run with 4 processors. As is evident, each phase of the calculation exhibits its own pattern of machine activity, so that the behavior during the timestep is quite complex. This pattern is recognizably similar for all multiprocessor SIMPLE runs we have performed, regardless of size or number of processors.

It is interesting to compare the machine activity during the heat conduction row sweep with that of the column sweep. The row sweep shows high crosstraffic loads and takes about 2.75 times as long as the column sweep. Both column and row sweeps show extensive use of processor local memory, mainly due to the local storage of coupling coefficients. In the light of the discussion above we expect this picture to change substantially for sufficiently large problems, which should show a less pronounced performance difference between column and row sweeps.

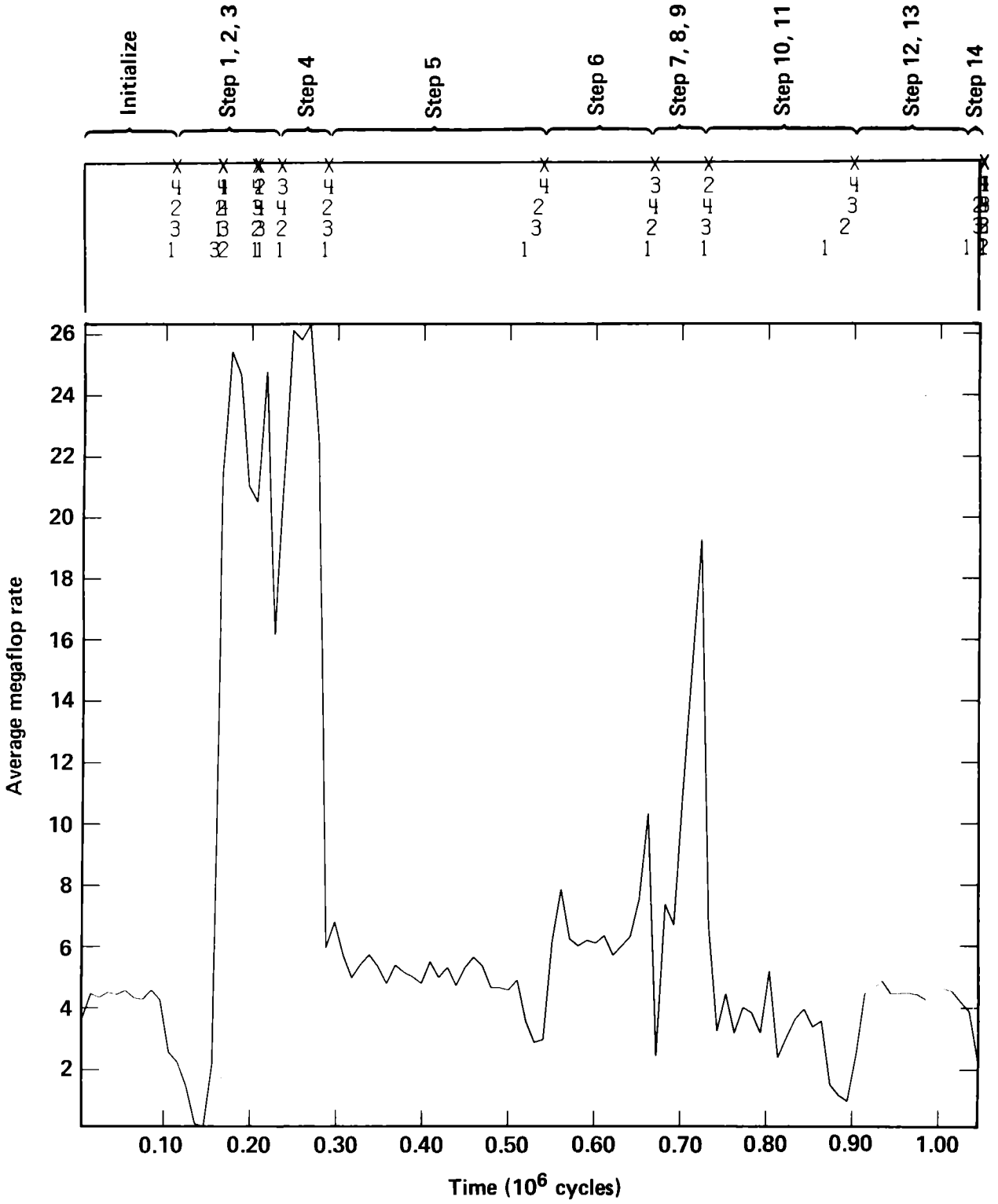


Figure 1

Simulated performance of 4 processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The upper portion of the figure shows the arrival times of the individual processors at the algorithm's synchronization points. All processors are restarted after the synchronization at the point marked "X". The step numbers refer to the calculational steps defined in the text. The lower portion of the figure shows the average per-processor megaflops vs. time measured in 25 ns ABOX cycles.

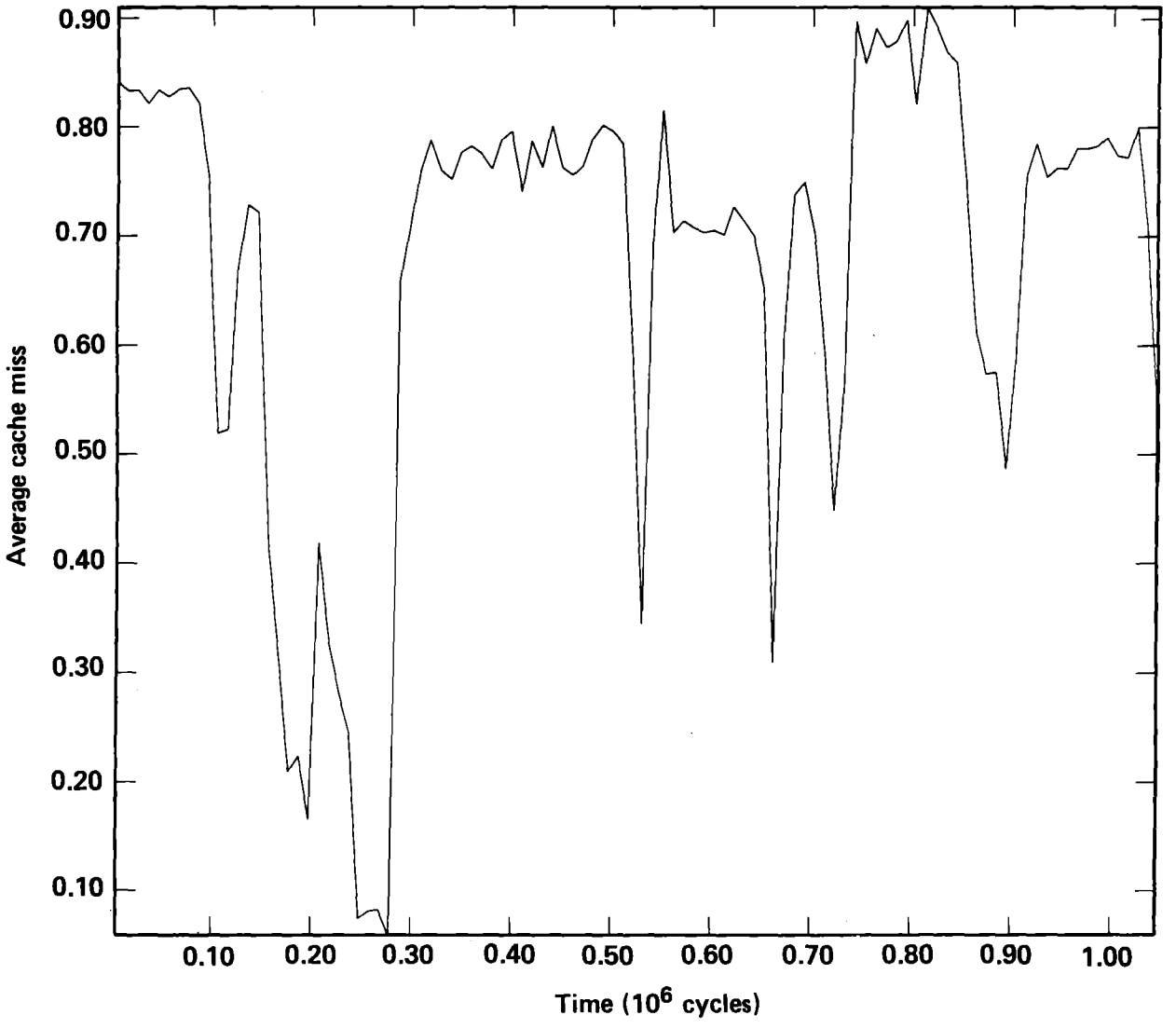


Figure 2

Simulated performance of 4 processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average fraction of time spent servicing data cache missed vs. time in 25 ns ABOX cycles.

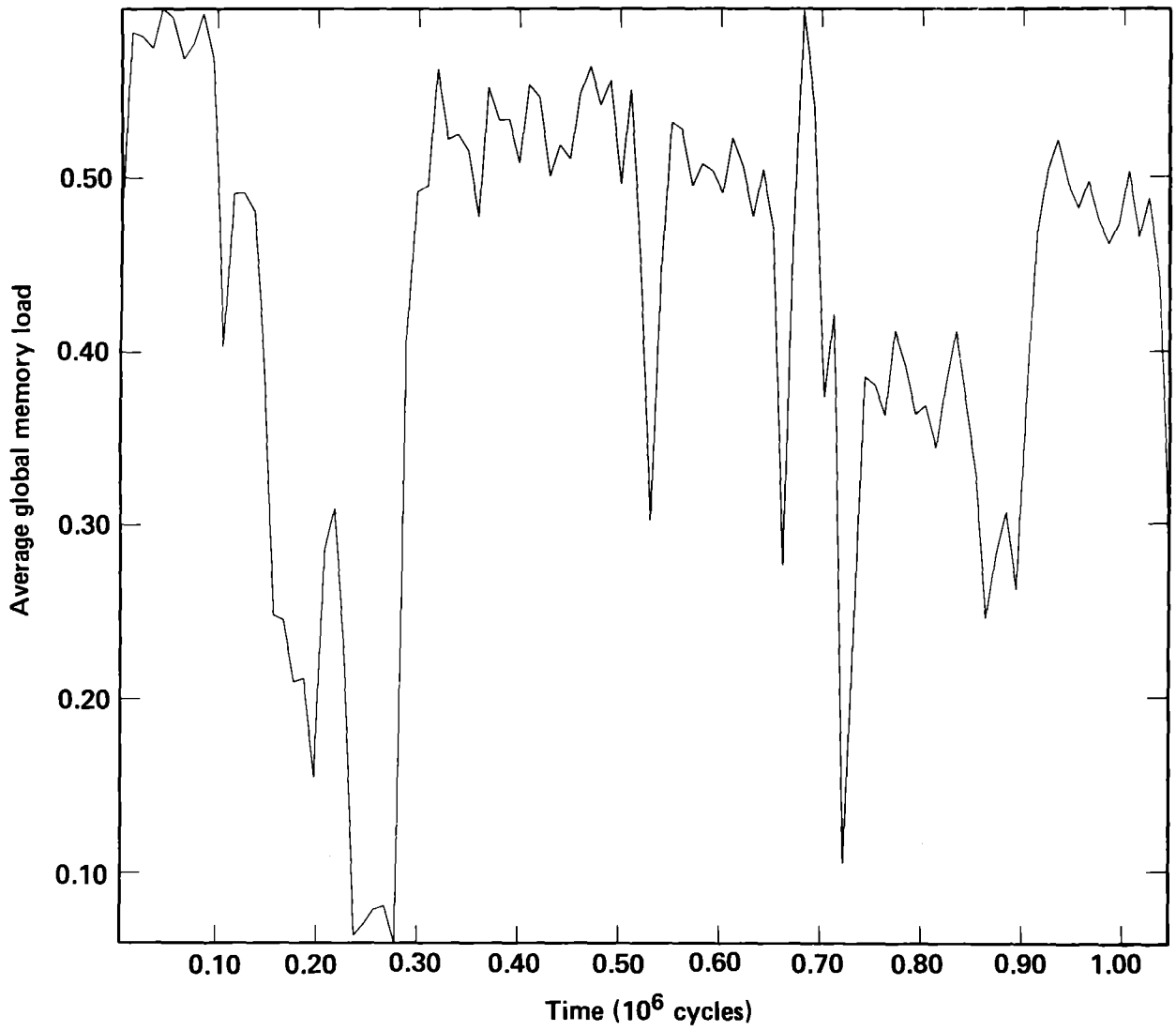


Figure 3

Simulated performance of 4 processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average global memory load as a fraction of total available bandwidth vs. time in 25 ns ABOX cycles.

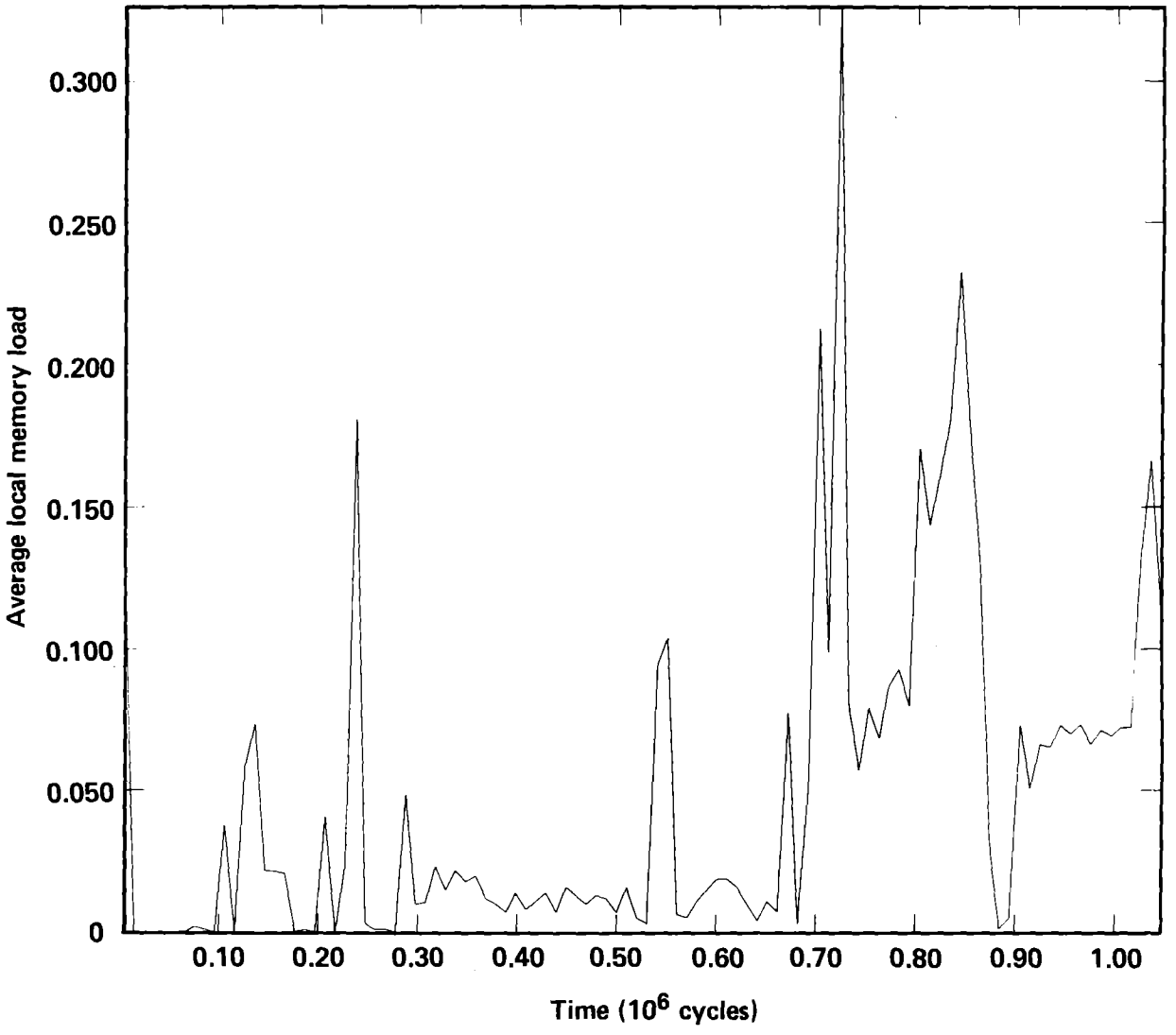


Figure 4

Simulated performance of 4 processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average local memory load as a fraction of total available bandwidth vs. time in 25 ns ABOX cycles.

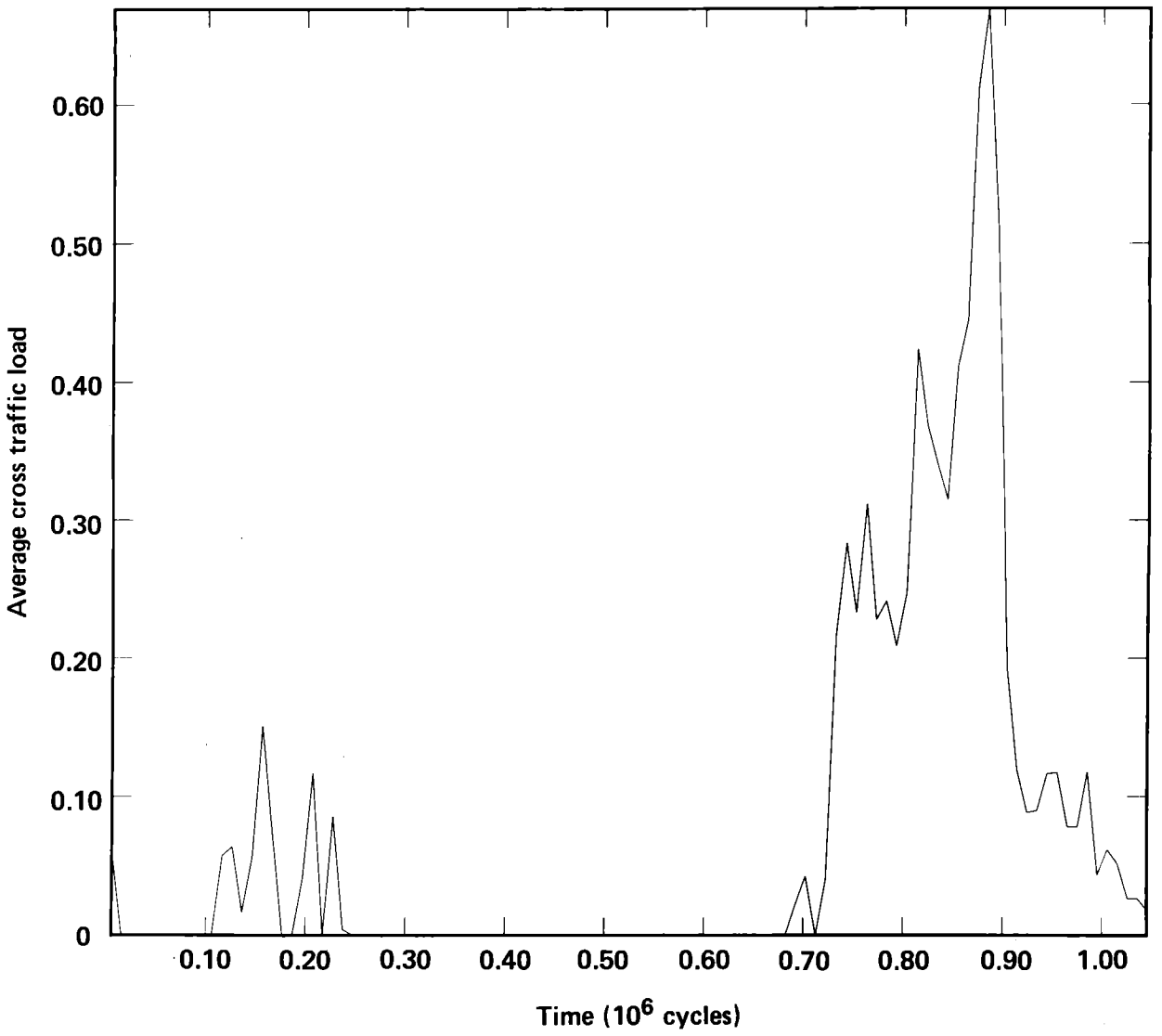


Figure 5

Simulated performance of 4 processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average fraction of time spent servicing interprocessor cache line transfer vs time in 25 ns ABOX cycles.

We can easily calculate the problem size for which this transition should occur. For the 90C,20R problem, each processor requires roughly 5*90*20/4 = 2250 distinct operands from the point it begins to access the shared temperature array during the column sweep until the column sweep is finished. This is substantially smaller than the cache size of 16384 words, so that each processor's entire column group is present in cache at the end of the column sweep. The subsequent row sweep then triggers the observed burst of cross traffic. For problems larger by a factor of roughly 8 (16000 zones), however, this situation will change and each processor at the end of the column sweep will already have started writing back to shared memory the first temperature elements it accessed.

In spite of its dramatic appearance, the row sweep is not the only cause of the inefficiency shown in Table 5. Analysis of the simulator runs allows us to assign the inefficiency to three major causes, as shown below.

	P=4	P=8
Waiting at synchronization barriers:	.03	.06
Global memory conflicts:	.15	.25
Interprocessor line transfer:	.06	.12
Total:	.24	.43

It is perhaps more useful to express these same numbers as "lost CPU's" by multiplying the fractional performance loss by P.

	P=4	P=8
Waiting at synchronization barriers:	.12	.48
Global memory conflicts:	.60	2.00
Interprocessor line transfer:	.24	.96
Total:	.96	3.44

We clearly must consider how this picture will change when the mesh size is greatly increased. As argued above, the fractional cost of both synchronization waiting and interprocessor line transfer should drop steadily with increasing mesh size, leaving global memory conflicts as the principal cost of multiprocessing. Simple models of multiprocessor memories [Yen82] predict that crossbar systems with equal numbers of processors and memories show an inefficiency due to conflicts that is nearly independent of P when P is greater than about 8. These facts taken together imply that efficiencies for sufficiently large problems should be fairly high (about 0.7) even for large numbers of processors.

This is not the end of the story, however. We must note that high efficiency does not necessarily imply high performance! It merely means that performance continues to grow linearly as processors are added. On a cache based machine, such as the S-1, the performance of each uniprocessor drops as problem size is increased. Table 6 shows the effect of varying the mesh size for SIMPLE in the single processor case.

Table 6

C	R	C*R	Mflops	Hit Ratio	Tfc Ratio
10	20	200	13.61	.9956	.078
15	20	300	12.68	.9952	.087
30	20	600	11.05	.9943	.108
60	20	1200	9.89	.9934	.128
90	20	1800	9.02	.9925	.145

On may expect that this decline will continue as problem size is further increased. The asymptotic value is difficult to predict without detailed analysis of the algorithm. A lower bound on performance is probably given by the results of Table 4, which would imply that programs dominated by long dyadic vector operations would have asymptotic performance of about 2 MFLOPS.

The issue of performance scaling with problem size therefore becomes complex. As problem size increases performance tends to also increase, due to the decreasing relative cost of synchronization waiting and interprocessor communication. At the same time, however, performance is negatively affected by the decreasing effectiveness of cache as the data set size increases.

3. The Biconjugate Gradient Algorithm

The biconjugate gradient algorithm (BCG) is a relatively new algorithm [Saa80] for the iterative solution of nonsymmetric systems of linear equations. Such systems are beginning to arise more often in applications and there is a dearth of methods for their solution. We had a chance to kill two birds with one stone by studying the behavior of BCG on the S-1, since we would learn about the behavior of BCG and, due to the general similarity of BCG to other, more common methods such as Incomplete Cholesky Conjugate Gradient, we would learn about the behavior of the S-1 on LLNL's typical linear algebra problems.

We imagined an operator on a two-dimensional grid of dimensions $nk \times nl$. We assumed the operator was "nearly" a 5-point diffusion operator so that the matrix we solved was $I + \beta J$, where J is a block tridiagonal matrix with blocksize nk. The diagonal blocks are tridiagonal matrices with central diagonal values equal to 4. and each off-diagonal value equal to -1. plus or minus a random perturbation. The off-diagonal blocks are diagonal matrices with diagonal values equal to -1. plus or minus a random perturbation. In both cases the random perturbations are applied as separate perturbations applied to each element, not one fixed value applied to all elements.

The value β is imagined to include various method dependent constants times the time-step size. Increasing β makes the problem more difficult. We are not interested here in the performance of the algorithm, but to give the reader an idea it solved a system with $\beta = 1/10$, $nk = 60$, and $nl = 40$ in just eight iterations to a relative accuracy of 10^{-5}.

For the remainder, let us just imagine that we have to solve $Ax = b$ where A is a nonsymmetric "five-striped" matrix with non-zero diagonal. The biconjugate gradient algorithm with diagonal preconditioning can be stated as follows. Let $M = \mathrm{diag}(A)$.

(a) Let $x_0 = M^{-1}b$.

(b) Let $r_0 = b - Ax_0$.

(c) Let $r_0^* = p_0 = p_0^* = r_0$.

(d) Calculate $\alpha = \langle r_0 , r_0^* \rangle$

We refer to the above as the "initialization" phase.

The significant operations required in each iteration are:

(a) Multiply A by a vector.

(b) Multiply A^* by a vector.

(A^* means the transpose of A. Note however that we do not have to form A^*, merely multiply by it.)

(c) Calculate M^{-1} times a vector (twice)

(d) Calculate dot products of two vectors (twice).

(e) Update r, r^*, p, p^* and x .

The matrix M is the preconditioning matrix. Different choices of M lead to different convergence rates and different costs per iteration. The choice of M must balance the ease of solving linear equations of the form $Mx = y$ with the need for M to resemble A is some way, in the sense that $M^{-1}A$ have a smaller condition number than A.

If the preceding paragraph is Greek to the reader, never mind. The M we have chosen makes the task of calculating M^{-1} times a vector easy, yet for the matrices in question yields a good convergence rate.

The algorithm divides easily into parallel calculations, and as a bonus these calculations are vector operations for the most part. We calculate each nk long segment of the answer using vector operations, with nl such segments to calculate. One could do better on a machine like the Cray-1 by treating the five diagonals as units and using the methods of [Mad76] but transforming such a method to a multiprocessor would be more difficult. As it is, with up to nl processors, we simply give each processor its share of the nl segments to do.

Here then is the entire algorithm, with the dashed lines representing synchronization points. There are unfortunately quite a few of these, since at these points some processor is about to need the results of another processor.

Initialization

1. $b=M^{-1}b$ (b is input as initial rhs, output as solution x)

   $\alpha = 0$

   rdotrs = 0

---

2. r=Ab (initial guess at solution)

---

3. $r=b-M^{-1}r$

   p=r

   p*=r

   r*=r

   bnorm=0,rdotrs=0 (done by processor 1 only)

---

4. $\alpha=\langle b,b\rangle$

   $rdotrs=\langle r,r^*\rangle$

   These two dot products are done with a semaphored critical section as each processor adds its partial dot-product to the total. The grand totals had to be previously initialized to zero, which they were in step 1. The same procedure is used in the iteration phase as noted below.

---

5. $bnorm=epsilon\cdot\sqrt{\alpha}$ (used to decide when to stop iteration)

   epsilon is an input, user controlled parameter, say 10^{-5}.

Iteration

6. $ap=Ap$
7. $t=M^{-1}p^*$

---

8. $ap=M^{-1}ap$
9. $asps=A^* t$

   $\alpha=0$ (done by processor 1 only)

---

10. $\alpha = \langle ap,p\cdot\rangle$

    Critical section for summing partial sums (see remark under 4 above).

---

11. if $\alpha = 0$ take error exit

    α = rdotrs/α

12. b = b + α • p

    r = r - α • ap

    r* = r* - α • asps

---

13. rnold=rdotrs, rdotrs = 0, deltax = 0 (done by processor 1 only)

---

14. deltax = <p,p>

    rdotrs = <r,r*>

    Critical section while global rdotrs, deltax summed from local parts

---

15. deltax = | α | $\sqrt{}$ deltax

    If deltax < bnorm then exit iteration

16. β=rdotrs/rnold

17. p = r + β • p

    p* = r* + β • p*

---

18. Iterate to step 6.

We simulated various S-1 configurations on various size problems with quite surprising results.

First we considered a medium sized problem with nk = 60, nl = 40 , and varied numbers of processors. The results are given in Table 7. We used double precision since typical LLNL matrices have large condition numbers. We restricted the iteration to two iterations rather than running to convergence, in order to conserve computer resources.

Table 7

Behavior of BCG algorithm with nk=60, nl=40, in double precision.

np	MFLOPS	Speed up over 1 processor	efficiency	Cache Hit Ratio	Traffic Ratio
1	2.6	1	1.0	.9785	.23
2	4.9	1.9	0.94	.9836	.17
4	14.5	5.6	1.4	.9911	.09
8	39.4	15.1	1.9	.9944	.05
16	47.6	18.3	1.14	.9942	.06

One Cray-1=32.0 MFLOPS.

There is a lot to say about Table 7. Right off the bat we seem to be violating an "obvious" principal of multiprocessing, namely that the efficiency, the speed using n processors divided by the quantity n times the speed of one processor, should be less than or equal to 1 . But using eight processors gives us an efficiency near 2 ! Isn't this impossible?

Clearly what must be going on is that something is slowing the one processor case down. A glance at the cache hit ratio reveals the problem. With just one processor, that processor has too much data for which it is responsible, and the data won't fit in the cache. Only 97.8% of the time is the desired line already in the cache, compared to 99.4% for the average of the eight processors.

Well, 97.8% sounds good but it isn't. This translates into a traffic ratio of .23, compared to only .05 for the eight processor case.

To test this hypothesis, we ran different sized problems on one processor. See Table 8.

Table 8

Behavior of BCG on an S-1 uniprocessor with varying problem sizes. Performance declines markedly with increasing problem size due to a declining cache hit ratio.

Problem size	MFLOPS	Hit ratio	Tfc ratio
10 x 40	9.6	.9957	.04
20 x 40	4.7	.9887	.12
40 x 40	3.2	.9828	.18
60 x 40	2.6	.9785	.23

A problem uses 14 vectors of length nk x nl , so in double precision the data occupy 28•nk•nl works. Thus a 20 x 40 problem should use just about 22K words, exceeding the cache size of 16K, while a 10 x 40 problem will fit easily.

For additional confirmation, we ran the problems in single precision and the results in Table 9 confirm the "16k barrier" effect. Note that 28•30•40=17600, and in the single precision column we are just starting to see the effect on the 30 by 40 problem.

Table 9

Performance of BCG on an S-1 uniprocessor in both single and double precision. When the data in use exceeds 16K words, performance begins to decline.

Problem size	MFLOPS Double Precision	MFLOPS Single Precision
10 x 40	9.6	18.2
20 x 40	4.7	19.1
30 x 40	---	13.6
40 x 40	3.2	9.2
60 x 40	2.6	7.0

It seems to us that one must draw a fairly ominous conclusion from this. A large 2-D problem in a hydrocode typically has a mesh size over 10K, and easily has 10-20 vectors of data in use. This means that even 8 processors will not be able to keep the problem in cache. If the results then degrade as they did in this problem, the results will be very poor.

Figure 6 shows the output of the simulator which plots MFLOP rate versus clock cycle. We have added to this figure indications of the location of each event in the algorithm list.

Figure 7 shows the output of the simulator for the same problem with 8 processors.

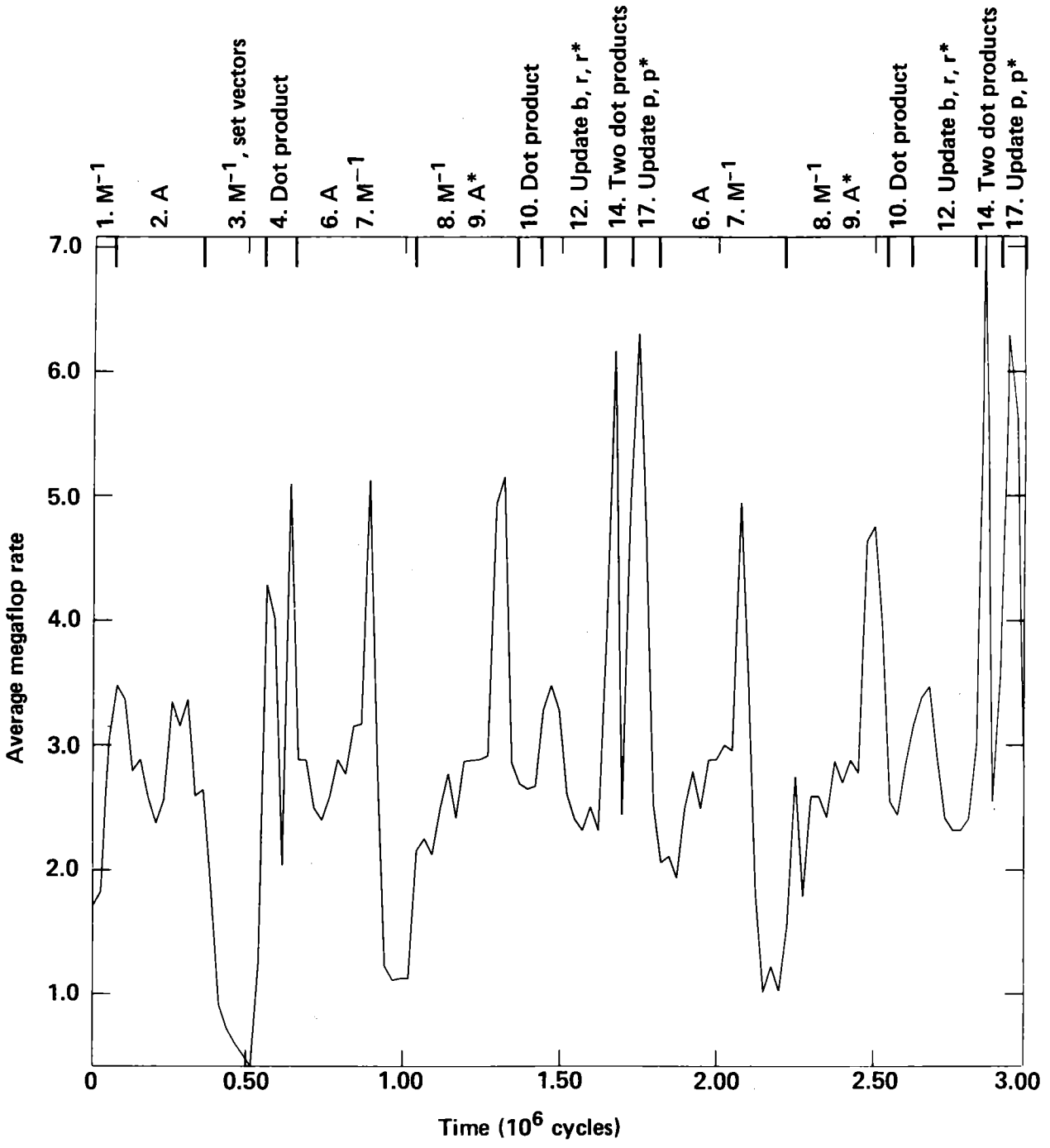


Figure 6

Average MFLOP rate per processor versus clock cycle for the initialization and first two iterations of BCG on a uniprocessor S-1, problem size nk = 60, n1 = 40, double precision. Shown above are the events occuring between the synchronization points, with events number keyed to the algorithm listing given previously. The synchronization after step 13 occurs so soon after step 12 that it cannot be distinguished on this plot.

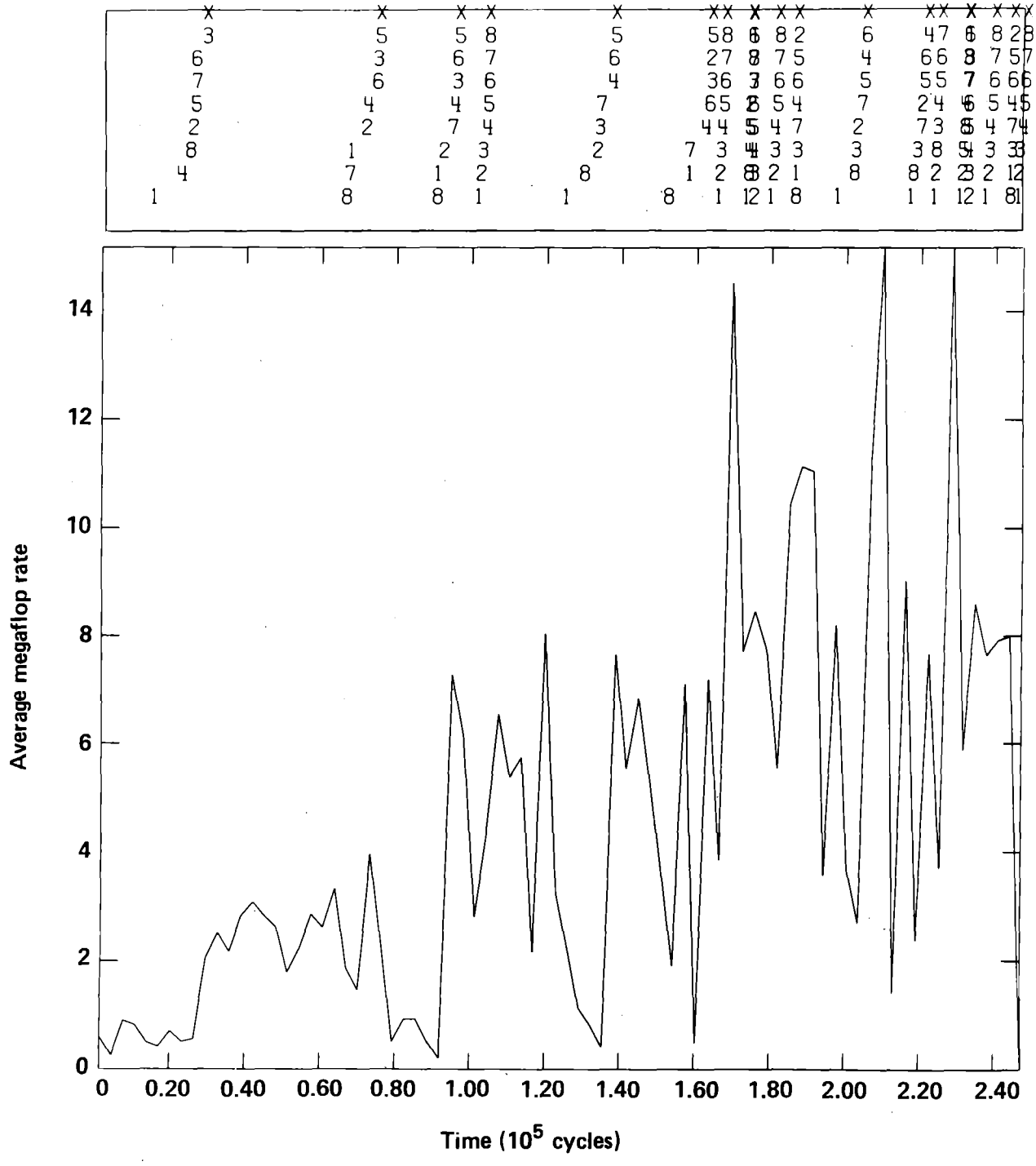


Figure 7

Average MFLOP rate per processor versus clock cycle for the initialization and first two iterations of BCG on an eight processor S-1, problem size nk = 60, nl = 40, double precision. The numbers above indicate the arrival of the different processors at each synchronization point. Compare with Figure 6.

Turning again to Table 7, let us consider the multiprocessor performance. This algorithm has little inherent cross traffic. There are five vectors (p, p*, r, r*, b) which get updated. Each of these is updated in sections, with each processor doing a section. Other processors may need to use the portions of a vector held by another processor when multiplying by A or A*, but this only involves the vectors p, p*, and the temporary vector t. This occurs with the multiplying by A or A* because of the outside blocks that exist in A's block tridiagonal structure. Since each of the processors is doing several nk long blocks, this sharing of p and p* will occur only at the boundaries between two processors. Therefore we can expect some increasing interference as the number of processors rises.

There is another "boundary effect" which we call "accidental line sharing". The 16 word line boundaries fall in the data more or less at random in our 60 by 40 problem. Therefore, at places in the vectors where a processor boundary occurs, data for two different processors may fall in the same line, so that conflicts occur in updating this line despite the fact that there is no theoretical conflict. This conflict could be avoided altogether by suitable overdimensioning of the matrices and vectors so that unused words are inserted after each nk block of data.

We did not do this, and did not do other things that the reader may notice, which may have increased performance in some cases. To us, the point of the exercise is to find out what happens to our normal computational physics problems when solved in a straightforward way. We have no doubt that the S-1, like every other machine, can be coaxed into much higher performance through the design of special algorithms, and the use special programming practices, especially assembly language coding. This has certainly been the case with our previous supercomputers. Our previous experience has also

taught us that this is a difficult job involving the special skills of some relatively rare individuals, and thus can be applied to only a few selected sections of major codes. We thus were interested in the "normal" performance of the machine on parallel and nonparallel algorithms, coded in a simple way in a higher level language, and sans any heroic measures.

For the 60 by 40 problem in double precision, we note little improvement when switching from 8 to 16 processors. Part of this is immediately attributable to an unequal work load, since 8 divides 40 evenly but 16 does not. The rest of the decline in efficiency is due to the two types of boundary effects noted above.

4. A Monte Carlo Problem

A typical Monte Carlo code for say, calculating neutron transport, is a difficult calculation to understand. With its many scalar operations and memory references, it would overwhelm our simulator. Therefore we devised a simple code which nonetheless has several of the most important qualities of a Monte Carlo code:

(a) Unpredictable calculation time for a "particle", with the total calculation being the calculation of thousands of "particles".

(b) When a "particle" undergoes a "transition", a table lookup is used to obtain the result of a random selection from a probability distribution.

(c) At each "transition", a weight is calculated and "energy" is deposited in some component of an array.

The problem chosen is the solution of a linear system $(I-H)x = y$ by the adjoint Monte Carlo method, described in Hammersly [Ham65]. The portion of the problem timed in the simulator consists of calculating an array x of length $m = 50$ by Monte Carlo sampling 2000 times. First, x is initialized to zero. When we begin a sample, or "particle", we choose the value of a variable i, which we call the state variable, from the values $1,2,\ldots,m$. This is done by choosing a uniform random variable and performing table lookup on a cumulative probability distribution. For example, to have an equal probability of birth in each state, we have an array $pbirth(j)$, $j = 1,\ldots m$ where $pbirth(j) = j/m$. Given $r \in [0,1)$, a random variable, we find the least j_0 such that $r < pbirth(j_0)$. Then we set $i = j_0$ and say that the "particle" is "born" in state i. A "score" is added to $x(i)$.

Next we determine the next "state" of the "particle" ℓ by using a $m \times m$ matrix C of "transition probabilities", where

$C(j,i) = \text{Prob}(\ell \leq j \text{ given } i)$.

That is, each column of C represents the cumulative probabilities for transitions from state i to other states. However, $C(m,i) < 1$ and the particle may "die" with probability $1-C(m,i)$. In this case we are done with this "particle". Assuming a new state ℓ is chosen, we score by adding a quantity to $x(\ell)$ and continue.

The matrix C is oerived from the matrix H as described in [Ham65]. We assume this has been done beforehand. There are restrictions on H which must be met in order to use this method to solve the linear system.

To summarize, the process consists of

(a) Birth

-- Choose i by lookup of random variable in table pbirth.

-- Calculate weight w

-- $x(i) = x(i) + w$

(b) Transitions until death occurs

-- Choose ℓ by lookup of random variable in $C(*,i)$

Possible stop here if random number $\geq C(m,i)$

-- Calculate weight w

-- $x(\ell) = x(\ell) + w$

-- Set $i = \ell$ and repeat (b)

This process is done independently some number of times. In all the runs reported here, this number is 2000 . The number of transitions/particle is a function of the matrix H. In our first examples the probability of a "death" averaged 1/2 , so that the expected number of transitions/particle should be

about 1 . (Even this strained our computer resources. We were able to make one run with more transitions which we will describe later. It took 30 minutes to run on the Cray simulator.)

We used a "vectorized" assembly language table lookup scheme called "33-section" and described in [Dub82c]. Using a standard FORTRAN bisection method resulted in so many memory references that the cost of a simulation run was unacceptably high. We did not trace the cost of the random number generator for similar reasons. Separate tests on the Cray indicate that this cost is about 1/3 of the other processes.

Exclusive of the random number calculations, the Cray performed the 342503 operations at a rate of 19 MFLOPS.

Table 10 shows the results of a straight transformation of the resulting code to the S-1 . The code itself is given in Table 11. Note the use of semaphores to protect elements of the array x from simultaneous updating.

Table 10

Performance of S-1 on Monte Carlo algorithm with various numbers of processors.

Number of processors	MFLOPS (Simulated)	Speedup	Efficiency
1	13.2	1.0	1.00
2	14.5	1.1	0.55
4	15.6	1.2	0.30
8	16.7	1.3	0.16
<Cray-1>	19.3		

Table 11

Code for Monte Carlo subroutine, showing use of semaphores to protect the elements of array x from simultaneous updating. The first section of code divides the work to be done into np equal parts. The "forkoff" statement creates the np processors. The statements psem(i) and vsem(i) set and clear, respectively, the i'th semaphore.

```
      Suoroutine mc(h,c,pbirth,tpb,x,a,m,npar)

c Adjoint form of monte carlo solution of linear equation (I-H)x=a
c Reference...Hammersly, Monte Carlo Methods
c Author:  P. F. Dubois
c The following three lines are cliches which expand into needed definitions
      use mpdefs
      use mpall
      use mploc
      dimension h(m,m), c(m,m), x(m), a(m), tpb(m), pbirth(m)
      data maxc/500/
c MULTIPROCESS---create np processors--prlcl is the processor number
      forkoff [np-1]
      nmin = 1+((prlcl-1)*npar)/np
      nmax = prlcl*npar/np
      mmin = 1+((prlcl-1)*m)/np
      nmax = prlcl*m/np
c initialize answer
      do 10 i = mmin, mmax
      x(i) = 0.
10    call vsem(i)
      call syncal
c loop on monte carlo tries
      do 20 ip = nmin, nmax
c find first state  using distribution tpb and random number generator ranf
c luf(x,table,m) returns index of x in table, m+1 if too big
      i = luf(ranf(0), tpb, m)
c calculate starting weight and score birth
      wg = a(i)/(pbirth(i)*npar)
      call psem(i)
      x(i) = x(i)+wg
      call vsem(i)
c collisional loop...change from state i to state 1
      do 30 icoll = 1, maxc
c find next state using i'th distribution
      1 = luf(ranf(0), c(1,i), m)
c was particle killed?
      if (1,gt,m) go to 20
c calculate current weight and score--real one in M.C. code would have more ops
      wg = wg*h(1,i)/abs(h(1,i))          ( so this is as clumsy as possible)
      call psem(1)
      x(1) = x(1)+wg
      call vsem(1)
c set i to new state and loop
      i = 1
30    continue
c error if we reach here--error coding omitted in this listing
20    continue
      call syncal
      return
      end
```

We studied the causes of this poor performance. The three problems that came to mind were:

(a) Unequal work performed by different processors

(b) Semaphore overhead

(c) Cross-traffic due to sharing array x between processors.

The databases C and pbirth are small and migrate gradually to cache. The plot in Figure 8 for 8 processors shows the MFLOP rate builds up while these arrays are being fetched, and then oscillates randomly around 16 MFLOPS with what one might interpret as a late burst of 18-19 MFLOPS just at the end. The marks show the finish time of each processor completing its 250 = 2000/8 particles. It is clear that the unequal work issue is not important here.

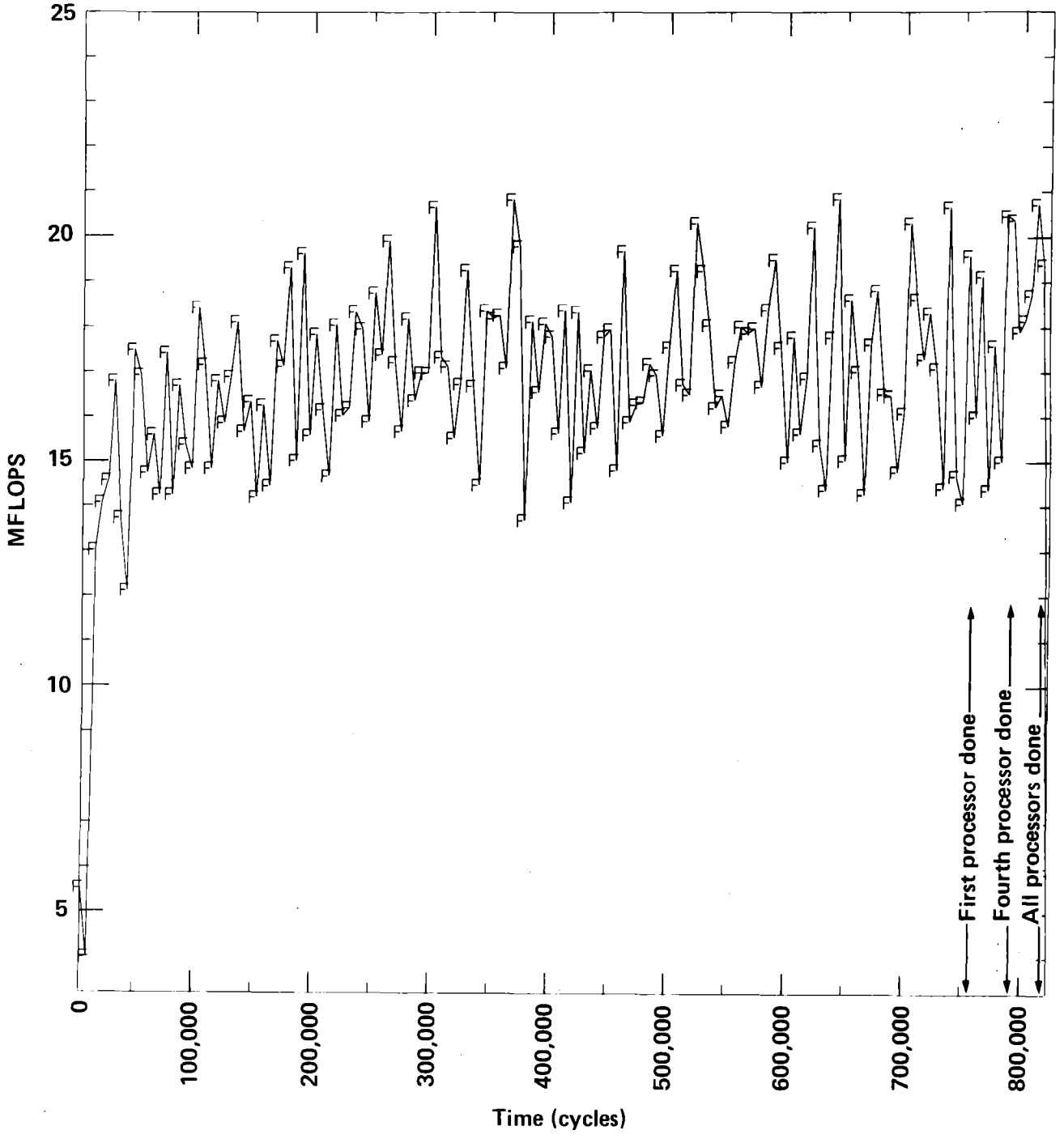


Figure 8

Total MFLOP rate on Monte Carlo benchmark for eight processors. Finish times for one, four and eight processors are marked. Note the startup period before full performance is achieved.

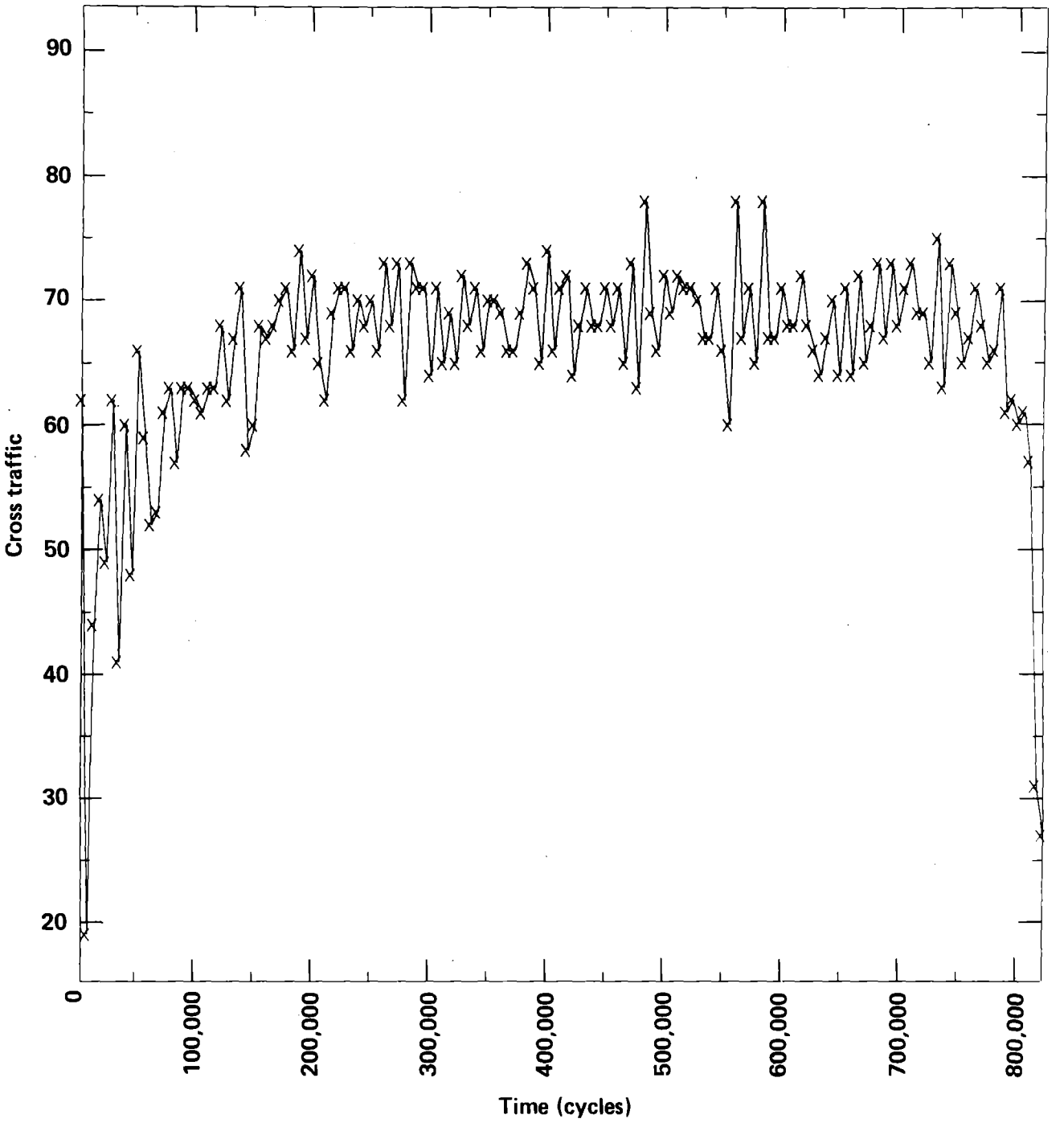


Figure 9

Cross-traffic rate between processors for the eight processor Monte Carlo problem. The cross traffic is very high as all processors update the answer array. Performance improves dramatically if this array is made local to each processor. Vertical axis is number of cross-traffic events in a 10000 cycle bin.

An investigation of the semaphore overhead question revealed that this problem was negligible, with almost no losses due to one processor having to wait for another to pass a critical section.

The startling thing in Figure 8 is that the MFLOP rate stays up after most of the processors have quit! Indeed, Figure 9 shows the cross-traffic between processors. The decline in cross traffic matches the decline in the number of processors still working, and the MFLOP perhaps even increases.

We thus predict that it is the sharing of the variable x which is responsible for the poor performance. To test this idea, we rewrote the code so that each processor accumulates into its own copy of the array x , and then these partial sums are accumulated at the very end. The results are shown in Table 12.

Table 12

Comparison of previous results with results using a local array for accumulating a portion of the answer, followed by summation of the partial results.

Number of Processors	MFLOPS (Global array)	MFLOPS (Local array)	Speedup vrs. global
2	14.5	26.1	1.8
4	15.6	47.9	3.1
8	16.7	75.7	4.5

This benchmark is very simple and of course does not reflect at all many important parts of a Monte Carlo code. The number of particles is arbitrary and not necessarily in the correct proportion relative to the cross-sections for interaction. It may be that the unequal work problem will be more or less important than shown here. Also, we may have overstated the S-1 performance since many of the operations are of the sort, like address calculations, which we do not simulate.

We ran one problem with a much higher number of transitions/particle for comparison. The simulator generated about 22 million words of data and used 30 minutes of Cray time for the calculation, giving the reader a hint at what simulating a real Monte Carlo code might amount to. The average MFLOP rate for eight processors rose to 99.5 MFLOPS. We found that with the larger amount of work to do, the proportion of the time spent getting up to speed is smaller, as is the proportion of the time lost to unequal work. The former was fairly significant in the simpler problem, representing inefficiencies which last until each processor has read into cache all of the cross-sectional data.

It is interesting to note, however, how much the performance is degraded during this time. This may imply, for example, that such a code doing timesharing with another code may be very much damaged, or that if you are going to do samples at all that you may as well do a lot, since the marginal cost of a sample falls off so dramatically. However, we are not certain on these points since our benchmark is so far from a real code. We can only treat these results as a tantalizing glimpse of the possibilities.

5. Finite Element Neutronics

Recently a new method of calculating time-independent neutron transport in one-dimension has been developed by J. Ferguson. The method is significantly faster than the S_n method. Anne Greenbaum's analysis of the method has classified it as a Petrov-Galerkin finite element method. Therefore we shall refer to the algorithm as the Petrov-Galerkin-Ferguson algorithm, or PGF for short. Studies of a time-dependent version of PGF are under way.

We tested PGF as is on the simulator. We then developed a parallel version of the algorithm, and tested it on simulated multiple processor machines. First we will explain the algorithm, and then given our results.

Let N be the number of radial points, $r_1, \ldots, r_N$. Let $K = 2M + 1$ be the number of angular points. The angles are represented as direction cosines relative to the radial axis, so that -1 represents flow toward $r = 0$ and 1 flow away from $r = 0$. We can imagine either spherical symmetry or plane symmetry; this only affects the boundary conditions.

We arrange our angular points from -1 to 1, so we have $-1 = \mu_1$, $\mu_2, \cdots \mu_M, \mu_{M+1} = 0, \mu_{M+2}, \cdots \mu_{2M+1}$. The left half of our computational mesh is shown in Figure 10.

The computation of the solution $f(i,j)$, $i=1,...N$; $j=1,...K$, proceeds in four steps:

(a) The first boundary condition

We calculate the left side of the upper boundary and the entire left edge, that is, $f(N,j)$, $j=1, \ldots, M+1$ and $f(i,1)$, $i=1, N-1$.

(b) The main calculation, left half.

We calculate the left $(\mu \leq 0)$ portion of the solution by proceeding down each column in turn, that is, $f(i,2)$, $i=N-1$ descending to $i=1$, followed by $f(i,3)$, $i=N-1$, 1 , etc., up to $f(i,m+1)$, $i=N-1, 1$.

(c) The second boundary condition.

The second boundary condition yields $f(1,j)$, $j=M+1, \ldots, K$, i.e. the rest of the bottom edge.

(d) The main calculation, right half.

We calculate the right side, this time moving up the columns.

This is shown more graphically in Figure 10. Please study this figure for a moment as we shall refer to it repeatedly in the sequel.

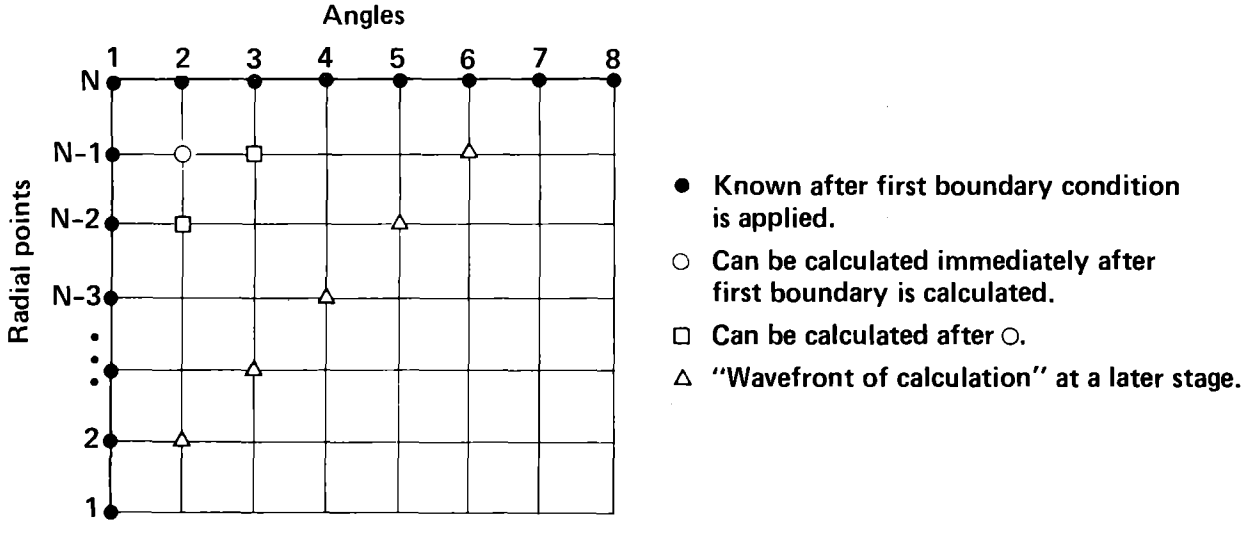


Figure 10

The left half of the computational grid, showing the locations known after applying the first boundary conditions to get the top and left edge (dark circles). The location circled is the first interior point to be calculated. The other marked points are referred to in the text.

The reason we calculate the values as we do is that for $\mu \leq 0$, that is, $j \leq M+1$, we can calculate a point $f(i,j)$ as soon as we know the 3 values at the corners of the rectangle of which $f(i,j)$ is the lower right corner:

```
f(i+1,j-1) ---- f(i+1,j)
    |               |
f(i,j-1)   ---- f(i,j)
```

During the calculation of $f(i,j)$ for $j > m+1$, we likewise must know values in the corners <u>below</u> and to the left of $f(i,j)$.

Other than knowing that $f(i,j)$ depends on these three values, the details of the calculation are not important to this discussion, except to say that the calculation involves roughly a hundred floating point operations and that external calls to source routines are required. We used very simple source routines for our tests. So we can say that $f(i,j)$ can be calculated when we know its neighbors, and without side effects, and that the calculation is substantial enough to keep the processor busy for a while. There is also some work in common that we could have extracted out and precalculated, but we did not do this in the interests of simplicity and in line with the philosophy we expressed earlier.

In Figure 10 , after calculating the upper boundary condition, we can solve an ODE along the line $\mu = -1$ to get values on the left edge. The value $f(N-1, 2)$ (shown circled) is the first interior point we can calculate, since we know its three neighbors.

Note that once this is done we could proceed to calculate $f(N-2, 2)$ <u>or</u> $f(N-1, 3)$ (Marked with a ⊔ in Figure 10). The scalar code proceeds in the former direction.

The parallel version of PGF uses a "wave-front" principle. We calculate a diagonal line of elements simultaneously. At stage 1 we calculate f(N-3, 2) . At state 2 we simultaneously calculate f(N-2,2) and f(N-1,3) . Next we do the three values f(N-3,2) , f(N-2, 3) and f(N-1, 2) . We continue to expand the number of processors in use until we either reach the column M+1 or all of the processors are in use. If there are m or more processors we reach all columns on one pass. Otherwise we need another pass or passes. With K = 17 , M = 8 , we can use up to 8 processors at once. In actual practice, such a seemingly small value as K = 17 is not uncommon at all. The spatial resolution is often considerably higher, with several hundred points not unheard of.

Naturally, when we reach the bottom of the column the number of processors in use drops gradually down to 1 again.

We do not have perfect parallelism here, but since N >> K in real problems, the penalty to build up our wave-front is not too severe. In fact, we can analyze just what it is.

Assume one processor can calculate one f(i,j) in time μ . The time for a scalar calculation, ignoring the boundary calculations, is thus:

$$t_s = (2M * (N-1))\mu .$$

Let the number of processors be P and assume P divides M evenly. The number of stages of calculation on the downward sweep is N-1+P-1 . This is done M/P times. Each stage takes μ units of time, for a parallel time for both downward and upward sweeps together of

$$t_p = 2(M/P)\ (N-1+P-1)\mu$$

Thus $t_s/t_p = P(N-1)/(N-1+P-1)$

The efficiency, $E = t_s/Pt_p = (N-1)/(N-1+P-1)$.

The efficiency of the calculation as a whole is reduced by the calculation of the left boundary, which cannot be done in parallel. A complete formulation is:

$$t_s = 2M * (N-1)\mu + (M+1)\mu_b + (N-1)\mu_\ell$$

$$t_p = 2[M/P](N-1+P-1)\mu + (M+1)\mu_b + (N-1)\mu_\ell$$

where μ_b is the time to calculate an upper boundary point and μ_ℓ is the time to calculate a left boundary point.

The relative values of μ, μ_b, and μ_ℓ depend on the actual coding for source terms, boundary conditions, etc. An actual measurement on the Cray-1 was made, yielding approximate relative values $\mu = 1$, $\mu_b = .1$, $\mu_\ell = .05$.

For $M = 8$, we get the following values of t_s/t_p for various N and P.

Table 13

Theoretical Values of t_s/t_p, $M = 8$

	N		
P	31	51	101
2	1.93	1.95	1.97
4	3.72	3.74	3.85
8	6.55	6.87	7.32

The corresponding efficiencies (t_s/Pt_p) are shown in Table 14.

Table 14

Theoretical efficiency, t_s/Pt_p, M = 8

	N		
P	31	51	101
2	.97	.98	.99
4	.93	.94	.96
8	.82	.86	.92

This, then, represents the inherent inefficiency of the algorithm. We cannot expect any multiprocessor to do better than this.

Scalar Results

We simulated the performance of a single S-1 processor. Our standard problem was N=51 , K=17 . We calculated a rate of 8.6 MFLOPS in single precision. The Cray-1 achieved 5.3 MFLOPS.

This impressive performance of the S-1 can be traced mostly to the cache. While the Cray is shipping results back and forth to memory, as it is instructed to do by any FORTRAN compiler, the S-1 keeps all the results in cache, ready to be used at the next stage. It also keeps such handy quantifiers as the radii and angle values in cache. It is able to do this because this one-dimensional test has a small amount of data. There is also an element of simulator optimism here too, since the address calculations for expressions like f(i-1,j+1) have been done for free.

Parallel Results

We recoded the algorithm for multiprocessors using the wave-front algorithm previously described. The same 17 x 51 test problem was used. The results are shown in Table 15, where we have included the theoretical efficiencies from Table 13 for convenience of comparison.

Table 15

Simulated S-1 performance on a 51 by 17 PGF test problem, showing actual multiprocessor speedups compared to those theoretically possible. S-1 MFLOPS based on the design speed.

P	MFLOPS	Actual Efficiency	Theoretical Efficiency	Actual/ Theoretical
1	7.6	1.0	1.00	1.00
2	9.3	.61	.98	.62
4	13.6	.45	.94	.48
8	17.6	.29	.86	.34

We can account for nearly every lost cycle in comparing the actual to the theoretical performance. Table 16 shows, for $P = 8$, the percentage of time each processor spent waiting for three different resources. These are:

(a) Synchronization, the processor was waiting for other processors to reach a sync point;

(b) Switch, the processors request for data was waiting for a switch path;

(c) Memory.

Table 16

Details of 8 processor PGF test. The columns show, for each processor, the cache miss rate, the percentage of its time that it spent waiting for, respectively, global memory, the interconnection switch, and for other processors to reach a synchronization point, and finally the average MFLOP rate for that processor.

Proc. no.	cache miss	wait for g mem.	wait for switch	wait for sync	megaflops
1	0.170	0.083	0.033	0.512	2.431
2	0.260	0.170	0.062	0.408	2.167
3	0.336	0.245	0.148	0.266	2.167
4	0.401	0.305	0.096	0.242	2.167
5	0.438	0.346	0.037	0.264	2.167
6	0.452	0.387	0.025	0.255	2.167
7	0.458	0.433	0.036	0.220	2.167
8	0.493	0.409	0.024	0.213	2.167
avg.	0.376	0.297	0.058	0.297	2.200

Total megaflops 17.600

If we sum these three waiting fractions for each processor we get the totals shown in Table 17.

Table 17

Total fraction of processor time spent waiting for global memory, interconnection switch, or synchronization.

Processor	Total wait time fraction
1	.628
2	.640
3	.659
4	.643
5	.647
6	.667
7	.689
8	.646
Average	.652

Note that each processor either had trouble waiting for data, or, when it didn't, had to wait for the others. (the Statistics for processor 1 are slightly different because this processor did the left boundary by itself). The average waiting time of .652 means an efficiency of about .35, nearly exactly what we measured as the ratio of actual to theoretical performance in this case.

Table 18 shows more detail for the $P = 8$ case. A variety of data like this, and informative plots, are produced by the simulator.

Table 18
Details of processor activity, memory bank loads, and processor memory loads for PGF problem with 8 processors.

Proc. no.	cycles busy	percent	gmem ref.	lmem ref.	cache ref.	m to others	m fr others	cache hit ratio
1	91246.0	39.48	227	25	21991	112	213	0.989
2	83691.0	36.21	229	23	19796	215	217	0.987
3	83875.0	36.29	230	23	19795	218	218	0.987
4	84099.0	36.39	233	23	19792	221	222	0.987
5	83955.0	36.33	231	23	19794	219	220	0.987
6	84317.0	36.48	234	23	19791	222	223	0.987
7	83313.0	36.05	232	23	19793	220	212	0.987
8	79789.0	34.52	176	23	19806	211	113	0.990

memory statistics load fractions

bank:	1	2	3	4	5	6	7	8
	0.020	0.216	0.391	0.000	0.000	0.000	0.000	0.000

memory statistics load fractions per processor

Proc. no.	cache	local	bank:	1	2	3	4	5	6	7	8
1	0.190	0.005		0.003	0.028	0.049	0.000	0.000	0.000	0.000	0.000
2	0.171	0.004		0.002	0.028	0.050	0.000	0.000	0.000	0.000	0.000
3	0.171	0.004		0.002	0.028	0.050	0.000	0.000	0.000	0.000	0.000
4	0.171	0.004		0.002	0.028	0.052	0.000	0.000	0.000	0.000	0.000
5	0.171	0.004		0.002	0.028	0.051	0.000	0.000	0.000	0.000	0.000
6	0.171	0.004		0.002	0.028	0.052	0.000	0.000	0.000	0.000	0.000
7	0.171	0.004		0.002	0.028	0.051	0.000	0.000	0.000	0.000	0.000
8	0.171	0.004		0.002	0.021	0.038	0.000	0.000	0.000	0.000	0.000
avg.	0.174	0.004		0.002	0.027	0.049	0.000	0.000	0.000	0.000	0.000

The reader may be wondering whether the all this traffic between processors is a real part of the algorithm or merely a detail of implementation that could have been eliminated. It is necessary. Please refer to Figure 10, and imagine that each processor is about to calculate one of the the values marked by triangles, say processor 1 doing the leftmost, processor 2 the next one, etc. Each processor has just finished calculating the value just above. Assuming the array is stored in the usual column order, the processor probably has the line containing that word, and the one it is about to calculate, in cache. Unfortunately, the next processor needs the previously calculated value too. If we store the data the other way, rowwise, then while the processor is calculating a value that will likely be in a new line, the neighboring processor still needs the value just calculated and that line is inevitably in the cache of the processor that just calculated it. If K is smaller than the line size there may be even more complicated relationships to consider, so the results we report above are for the data stored by columns, that is, with successive radial points adjacent.

It is worth mentioning that this calculation might be significantly speeded up in multiprocessor mode by use of the S-1 message passing hardware. Using this hardware, the new results needed by neighboring processors could be shipped directly to the processors that need them. This would dramatically reduce the problems we discussed above. Of course, it would also require a significant reprogramming effort. When an actual S-1 multiprocessor is available we look forward to trying this out.

Inspired by the fact that Table 18 shows all the data in the first three banks, we tried changing the page size to 32 words to see if performance could be improved in this case by spreading out the data through 8 banks. It did improve, but only to 19.2 MFLOPS from 17.6 MFLOPS.

We also ran several cases in double precision. There was virtually no degradation of performance. For example, for P = 8 we got 17.4 MFLOPS.

Our colleague Anne Greenbaum tried a parallel version of the algorithm on the HEP (uniprocessor) computer. The HEP was very much slower than either the Cray-1 or S-1. Interestingly, though, it did achieve nearly the full theoretical speedup using the multiple processes/processor features of the HEP. It was also easy to express the desired concepts in the HEP's extended FORTRAN. We have not had a chance to try a multiple PEM version of the HEP, that is, with multiple processes per processor and more than one processor. The mind reels to think of the number of different algorithms there are to try on such a system.

We conclude that while a single processor will do well on this problem since the data fits in cache, the need to share just completed results significantly degrades the performance of the S-1 below that predicted by theory, or that achievable with a multiple processes per processor architecture. However, this could be improved with reprogramming to use the message passing hardware.

V. CONCLUSION

The simulations we have reported on were undertaken with the goals of investigating the performance issues raised by MIMD machines and gaining experience with the programming techniques required to utilize them. As yet we have explored only a limited set of algorithms and a single simulated machine. As discussed earlier, we have taken existing uniprocessor algorithms and extended them to a shared memory multiprocessor with minimal changes. Clearly future algorithms may depart radically from this approach. Our simulations are also deficient in that real problems run on fast multiprocessors will in general have many more zones than those we have been able to treat.

In view of all these limitations, what have we actually learned? Our experience to data with the simulator allows us to draw three tentative conclusions about the use of MIMD machines for solving large scientific problems:

1. The S-1 can be used as a multiprocessor in two relatively distinct modes. These are a shared memory, or tightly coupled, approach in which problem data is primarily stored in shared memory; and a distributed processing, or loosely coupled, approach in which problem data is primarily stored in private memory and communication takes place when required through the interprocessor message network. The shared memory approach, which we have used here, is relatively simple to program using a few extensions to FORTRAN. The distributed computing approach appears to offer higher performance for many algorithms. However, substantial programming effort and significant language extensions would be required to realize this potential.

2. Extrapolation of our simulator results to much larger problems indicates that many of the factors which limited our speedups in the $4 < P < 16$ range will be greatly reduced in importance. This is particularly true for:

a. Overhead operations which result from partitioning the algorithm. These include process management operations and communication between processes.

b. Synchronization penalty that results from speed variation between processes.

On the other hand, conflicts between processors in obtaining access to shared resources (usually memory banks and communication paths) will continue to be important. In considering the scaling issue, the effect of cache memory may outweigh all of these factors, however, which brings us to our final point.

3. The usefulness of a traditional data cache for scientific problems with large data sets appears questionable. In the cases we have studied, performance drops rapidly with increasing problem size. Clearly this performance drop can be delayed with algorithms which optimize cache hit rates. It does not appear feasible to do this by hand, however, except for simple cases. Compilers clearly must perform this tasks if it is to be done at all. The problem is difficult since the optimization approach must be dependent on data set size.

In the future we plan to further increase our understanding of MIMD machines and algorithms by pursuing the comparison of our simulator results with actual measurements on the S-1 MkIIa multiprocessor, when it becomes available. Additionally, we feel that improved analytic models for MIMD performance can be constructed which will be quite useful. The work of Briggs [Bri80], Dubois [Dub82a,b], Yen et al [Yen82], and Gilbert [Gil79], among others, provide an excellent foundation on which to build them.

References

[Axe82] Axelrod, T. S., Chase, L., Eltgroth, P. G., and Sloan, L. S., "Multiprocessor Simulator User's Manual," Lawrence Livermore National Laboratory, Livermore, Calif., UCID-19594 (1982).

[Bri80] Briggs, F. A. and Dubois, M., "Modeling of Synchronized Iterative Algorithms for Multiprocessors", Proceedings of the 18th Annual Allerton Conference, 1980.

[Cox78] Cox, L. A., "Performance Prediciton of Computer Architectures Operating on Linear Mathematical Models", Lawrence Livermore Laboratory UCRL-52582, 1978.

[Cra76] CRAY-1 Computer System Reference Manual, Cray Research Inc. Publication 2240004, 1976.

[Cro78] Crowley, W. P., Henorikson, C. P., and Rudy, T. E., Lawrence Livermore Laboratory UCID-17715, 1978.

[Dub82a] Dubois, M., and Briggs, F. A., "Performance of Synchronized Iterative Processes in Multiprocessor Systems", IEEE Trans. Software Eng., Vol. SE-8, No. 4, p.419-431, 1982.

[Dub82b] Dubois, M., and Griggs, F. A., "Effects of Cache Coherency in Multiprocessors", IEEE Trans. Computers, Vol. C-31, p.1083, 1982.

[Dub82c] Dubois, P. F., "Swimming Upstream: Calculating Table Lookups and Piecewise Functions", in Rodrigue, ed., Parallel Computations, Academic Press, New York, 1982.

[Far80] Farmwald, P. M., Bryson, W., and Manferdelli, J. L., "Signal Processing Aspects of the S-1 Multiprocessor Project", Proc. Soc. Photo-Opt. Instrum. Eng., Vol. 241, p.224, 1980.

[Far81] Farmwald, P. M., "On the Design of High Performance Digital Arithmetic Units", Ph. D. thesis, Stanford University, 1981.

[Far82] Farmwald, P. M., Lawrence Livemore National Laboratory, Livermore, Calif., private communication (1982).

[Gil79] Gilbert, E. J., "Investigation of the Partitioning of Algorithms Across an MIMD Computing System", S-1 Project 1979 Annual Report, Lawrence Livermore Laboratory UCID-18619, Vol 1, 1979.

[Ham65] Hammersly, J. M. and D. C. Handscomb, Monte Carlo Methods, Methun & Co. Ltd., London, 1965.

[Kog81] Kogge, P. M., "The Architecture of Pipelined Computers", McGraw-Hill, 1981.

[Lor72] Lorin, H., "Parallelism in Hardware and Software", Prentice-Hall, Inc., 1972.

[Mad76] Madsen, N., G. Rodrigue, and J. Karush, "Matrix Multiplication by Diagonals on a Vector/Parallel Processor", Information Processing Letters, Vol. 5, No. 2, June 1976.

[S-179] Wood, L. L., ed., "S-1 Project 1979 Annual Report", Lawrence Livermore Laboratory UCID-18619 (3 vols), 1979.

[Saa80] Saad, Y., "The Lanczos Biorthogonalization Algorithm and Other Oblique Projection Methods for Solving Large Unsymmetric Systems", UIUCDCS-R-80-1036, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1980.

[Smi78] Smith, B. J., "A Pipelined, Shared Resource MIMD Computer", Proc. 1978 Int'l. Conf. Parallel Processing, Bellaire, MI, p.6, 1978.

[Yen82] Yen, D. W., Patel, J. H., and Davidson, E. S., "Memory Interfereence in Synchronous Multiprocessor Systems", IEEE Trans. Computers, Vol. C-31, p.1116, 1982.